

# Course 925

# SQL Programming

# Language Introduction

by  
**Dag Hoftun Knutsen**

Technical Editor:  
**Jason T. Archambeau**

# Copyright

© LEARNING TREE INTERNATIONAL, INC.

All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc. Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

# Introduction and Overview



# Course Objectives

---

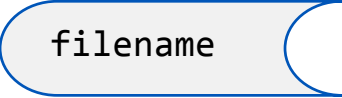
- ▶ **Use SQL to define and modify database structures**
  - CREATE, ALTER, DROP
- ▶ **Use SQL to update database contents**
  - INSERT, UPDATE, DELETE
  - Transactions, COMMIT, ROLLBACK
- ▶ **Query database content**
  - Simple SELECT
  - Joins
  - Functions
  - Aggregates
  - Subqueries
  - Views

SQL = structured query language



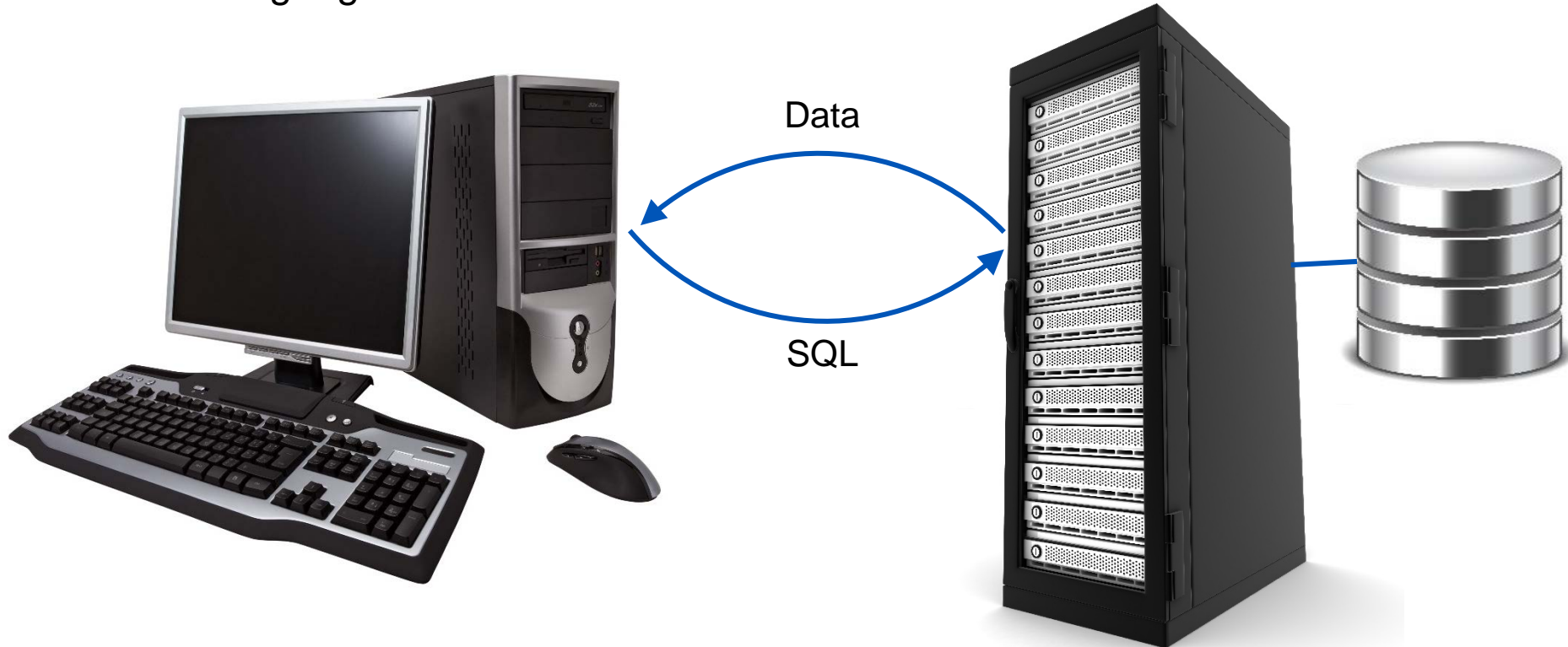
# Examples

---

- ▶ **The course contains many examples of queries**
  - Examples are shown and explained in the Course Notes
- ▶  **filename** indicates where the example can be found
- ▶ **The files are in the C:\Course925Examples folder**

# The Role of SQL

- ▶ **SQL is a common interface between client and database servers**
- ▶ **SQL is the interface between the application program and the database**
  - SQL is a data *sublanguage*
    - Which is more like an access method than a complete programming language



# The Role of SQL

---

- ▶ **Application programs may be**
  - 3GL programs
  - 4GL or application generator programs
  - Report generator programs
  - End-user point-and-click programs
  - Spreadsheets
  - Any frontend tool with SQL interface capability
  - Stored procedures or triggers

3GL = third-generation language  
4GL = fourth-generation language



# Result-Oriented

## ► SQL is a *result-oriented* language

- Specifies the desired *result* rather than step-by-step instructions of what to do

## ► Example

- If we have this table:
- Then this query:

c0-01.sql

- SELECT FirstName, LastName
- FROM Employees
- WHERE CurrentSalary >= 6000;

- Will produce this result:
  - Notice that the result is also on table format

EmployeeID	LastName	FirstName	CurrentSalary
1	Davolio	Nancy	2000
2	Fuller	Andrew	9000
3	Leverling	Janet	6000
4	Peacock	Margaret	3000
5	Buchanan	Steven	5000
6	Suyama	Michael	3600
7	King	Robert	3000
8	Callahan	Laura	4000
9	Dodsworth	Anne	5200

FirstName	LastName
Andrew	Fuller
Janet	Leverling

## ► SELECT statements will be covered in detail



# SQL Standard

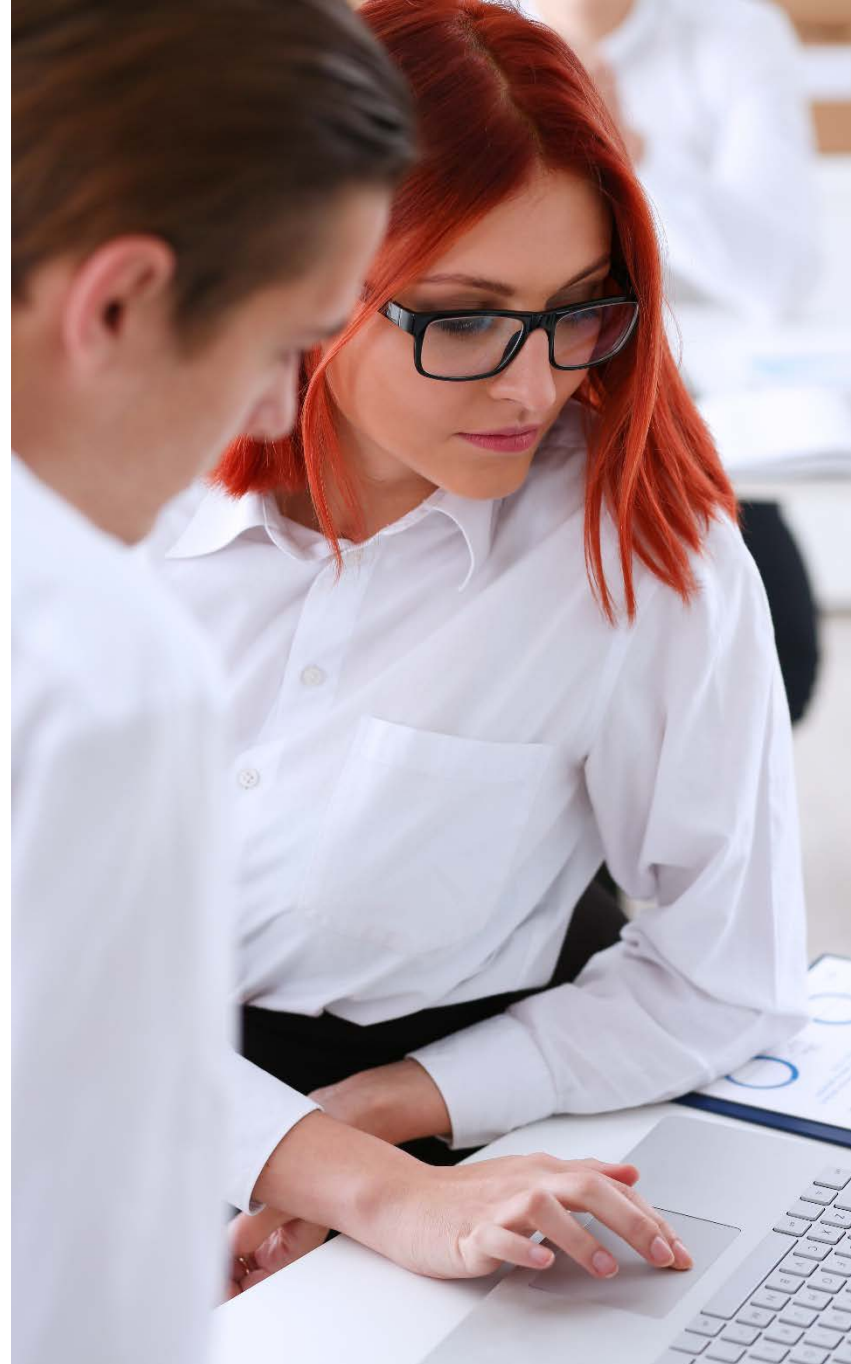
---

- ▶ **All relational database management systems use SQL**
  - Unfortunately, different products have different dialects of it
  - There is an ANSI/ISO standard that most products follow to a certain extent
    - SQL2 (also called SQL-92) is currently the most widely adopted standard
    - The standard was expanded in 1999 and 2003
- ▶ **This course will concentrate on the standard, but will also point out where popular products deviate from the standard**
  - PostgreSQL
  - Oracle
  - SQL Server
- ▶ **We will use PostgreSQL and SQL Server as basis for the Hands-On Exercises**
  - Product-specific features will be clearly identified

# About the Hands-On Exercises

---

- ▶ **In the exercises, you will choose between working in the PostgreSQL or the SQL Server environment**
  - Solutions are provided as files as well as in the Exercise Manual
  - Most solutions are product independent
  - In the few situations where the products differ, separate solutions are provided



# Related Courses

---

**Learning Tree provides a number of database-related courses**

- ▶ **Microsoft SQL Server courses covering various versions**
- ▶ **Oracle courses covering various versions**
- ▶ **Product-independent courses on database design techniques and more advanced SQL**

# Course Contents

---

## **Introduction and Overview**

<b>Chapter 1</b>	<b>SQL Fundamentals</b>
<b>Chapter 2</b>	<b>Data Definition</b>
<b>Chapter 3</b>	<b>Data Modification</b>
<b>Chapter 4</b>	<b>Single-Table SELECT Statements</b>
<b>Chapter 5</b>	<b>Joins</b>
<b>Chapter 6</b>	<b>Set Operators</b>
<b>Chapter 7</b>	<b>Row Functions</b>
<b>Chapter 8</b>	<b>Aggregate Functions</b>
<b>Chapter 9</b>	<b>Subqueries</b>
<b>Chapter 10</b>	<b>Views</b>
<b>Chapter 11</b>	<b>Course Summary</b>
	<b>Next Steps</b>



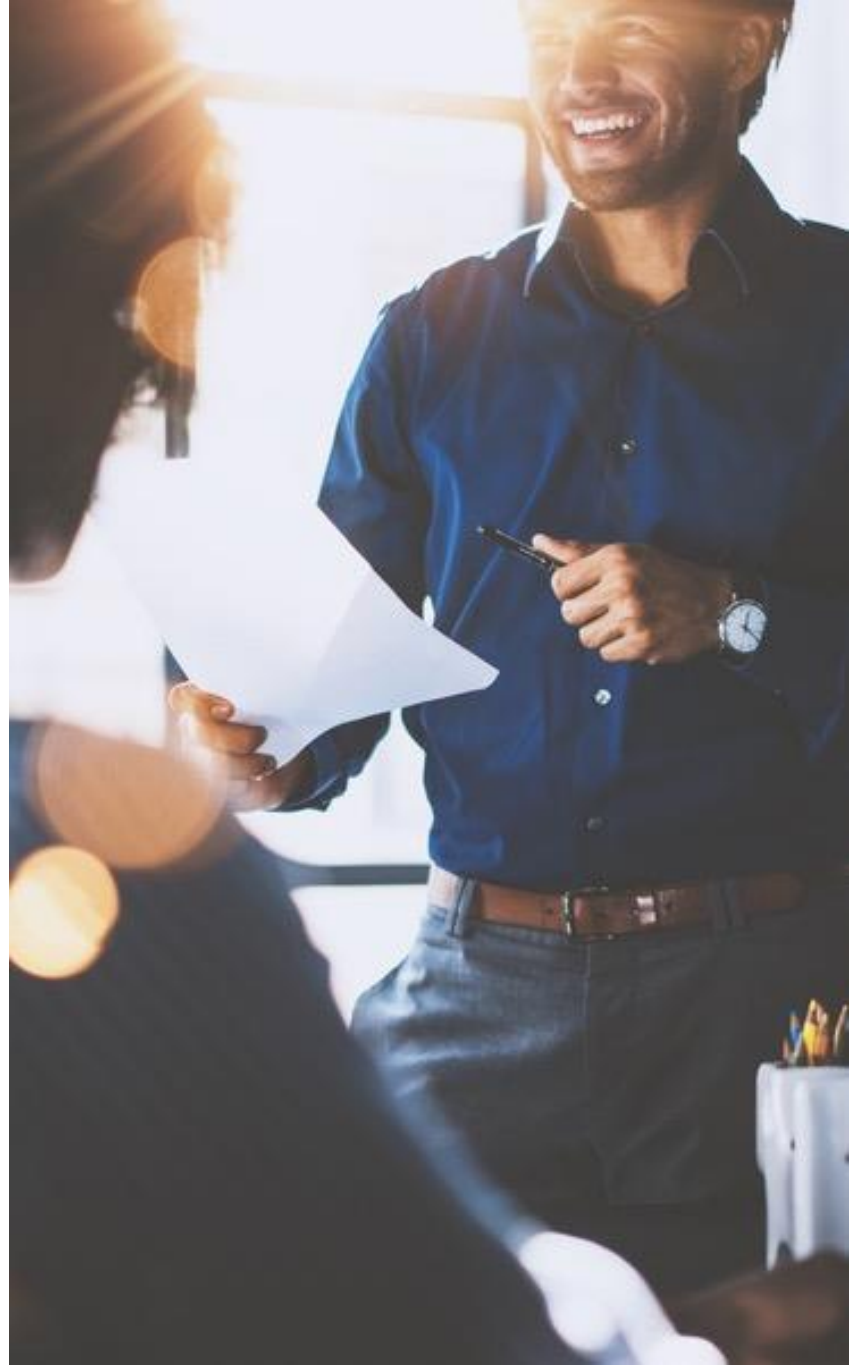
# Chapter 1

# SQL Fundamentals

# Objectives

---

- ▶ **Describe why SQL is important**
- ▶ **Review the fundamentals of database theory**
- ▶ **Review the evolution and history of SQL**
- ▶ **Describe the course environment**



# Contents

---

## The Importance of SQL

- ▶ Database Fundamentals
- ▶ History and Evolution
- ▶ Course Environment





# Why SQL Is Important

---

- ▶ **SQL is the only way to build, manipulate, and access a relational database**
  - There are several GUI tools available, but they all use SQL behind the scenes
- ▶ **There are GUI tools that can build SQL statements for you**
  - Those tools do not necessary build good SQL; those tools are rarely able to use the full power of SQL
- ▶ **Your ability to produce good database applications will be highly dependent on your knowledge of SQL**
  - There is no excuse for not learning SQL properly
  - This course provides a good foundation

GUI = graphical user interface





# Contents

---

- ▶ The Importance of SQL

## Database Fundamentals

- ▶ History and Evolution
- ▶ Course Environment



# Basic Building Block: Table

- ▶ A relational database consists of *tables*
- ▶ Each table has *columns* and *rows*
  - Each *column* is identified by its *column name*
    - Not by column position
  - Each *row* is uniquely identified by the *value(s)* in one or more columns
    - A unique column or combination of columns is called the *primary key*

Primary key



<u>ProductID</u>	ProductName	UnitPrice	CategoryID
1	Chai	18.00	1
2	Chang	19.00	1
3	Aniseed Syrup	10.00	2
4	Chef Anton's Cajun Seasoning	22.00	2
5	Chef Anton's Gumbo Mix	21.35	2
6	Grandma's Boysenberry Spread	25.00	2
7	Uncle Bob's Organic Dried Pears	30.00	7
...	...	...	...

# Relationships Between Tables

- ▶ **Tables are related via a column or a combination of columns that reference(s) the primary key of another table**
  - Such a reference is called a foreign key
  - A primary key/foreign key pair usually implies a one-to-many relationship

Primary key

Foreign key

Primary key

<u>ProductID</u>	ProductName	UnitPrice	CategoryID	<u>CategoryID</u>	CategoryName
1	Chai	18.00	1	1	Beverages
2	Chang	19.00	1	2	Condiments
3	Aniseed Syrup	10.00	2	3	Confections
4	Chef Anton's Cajun Seasoning	22.00	2	4	Dairy Products
5	Chef Anton's Gumbo Mix	21.35	2	5	Grains/Cereals
6	Grandma's Boysenberry Spread	25.00	2	6	Meat/Poultry
7	Uncle Bob's Organic Dried Pears	30.00	7	7	Produce
...	...	...	...	8	Seafood

# Normalization

---

- ▶ **A recommended database design principle says that tables should be on *third normal form (3NF)***
  - The principle is to have each table describe one thing only, rather than a mix of different things
- ▶ **Rule of thumb**
  - Each attribute should be dependent on
    - The key
    - The whole key
    - Nothing but the key
- ▶ **Example**
  - The example tables on the previous slide are on 3NF
  - It would have been possible to include CategoryName as a column in the Products table instead of having a separate Categories table, but that would violate the 3NF rule
    - CategoryName is dependent on CategoryID, which is not the primary key in the Products table
    - Products and Categories are different “things”



# Terminology

---

- ▶ **The relational database theory is based on mathematical theory, and the terminology is flavored by this**
  - Even if the mathematical terms are rarely used in daily speech, they are frequently used in academic textbooks
    - So it is useful to be familiar with them
- ▶ **Terminology**
  - A table is also called a *relation*
    - This is the origin of the term *relational database*
  - Rows are also called *tuples*
  - Columns are also called *attributes*
  - Primary key is also called *identifier* or *unique identifier*
  - Foreign key is also called *reference*
  - The number of columns in a table is called *degree* or *arity*
  - The number of rows in a table is called *cardinality*

# Contents

---

- ▶ The Importance of SQL
- ▶ Database Fundamentals

## History and Evolution

- ▶ Course Environment



# SQL History

---

- ▶ **1969: General idea described by Edgar (Ted) Codd, IBM**
  - A database system based on tables and relational algebra
- ▶ **1976: IBM prototype System R, origin of DB2**
  - Also the origin of SQL
- ▶ **1979: Oracle, first commercial database with SQL**
- ▶ **1986: First ANSI/ISO SQL standard**
- ▶ **1989: Addendum to ANSI/ISO standard**
  - Support for constraints
- ▶ **1992: ANSI/ISO SQL2 standard**
  - Which most products currently relate to
- ▶ **1999 and 2003: Additions to the ANSI/ISO standard**
  - Gradually adopted in products

# ANSI/ISO Standards

---

- ▶ **The ANSI/ISO SQL2 1992 Standard has three levels**
  - Entry level
    - Supported by most systems
  - Intermediate level and full level
    - Partly supported by many systems
    - Not fully supported by any system
- ▶ **The ANSI/ISO 1999 and 2003 Standards have two levels**
  - Core level
    - Basically similar to the previous entry level
    - Most products conform to this
  - Various extensions covering different areas
    - Some extensions implemented in some products
- ▶ **The standards are not complete, and all products have features that go beyond the standards**



# Contents

---

- ▶ The Importance of SQL
- ▶ Database Fundamentals
- ▶ History and Evolution

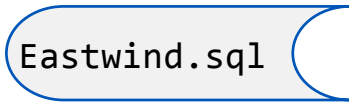
## Course Environment



# The Course Database

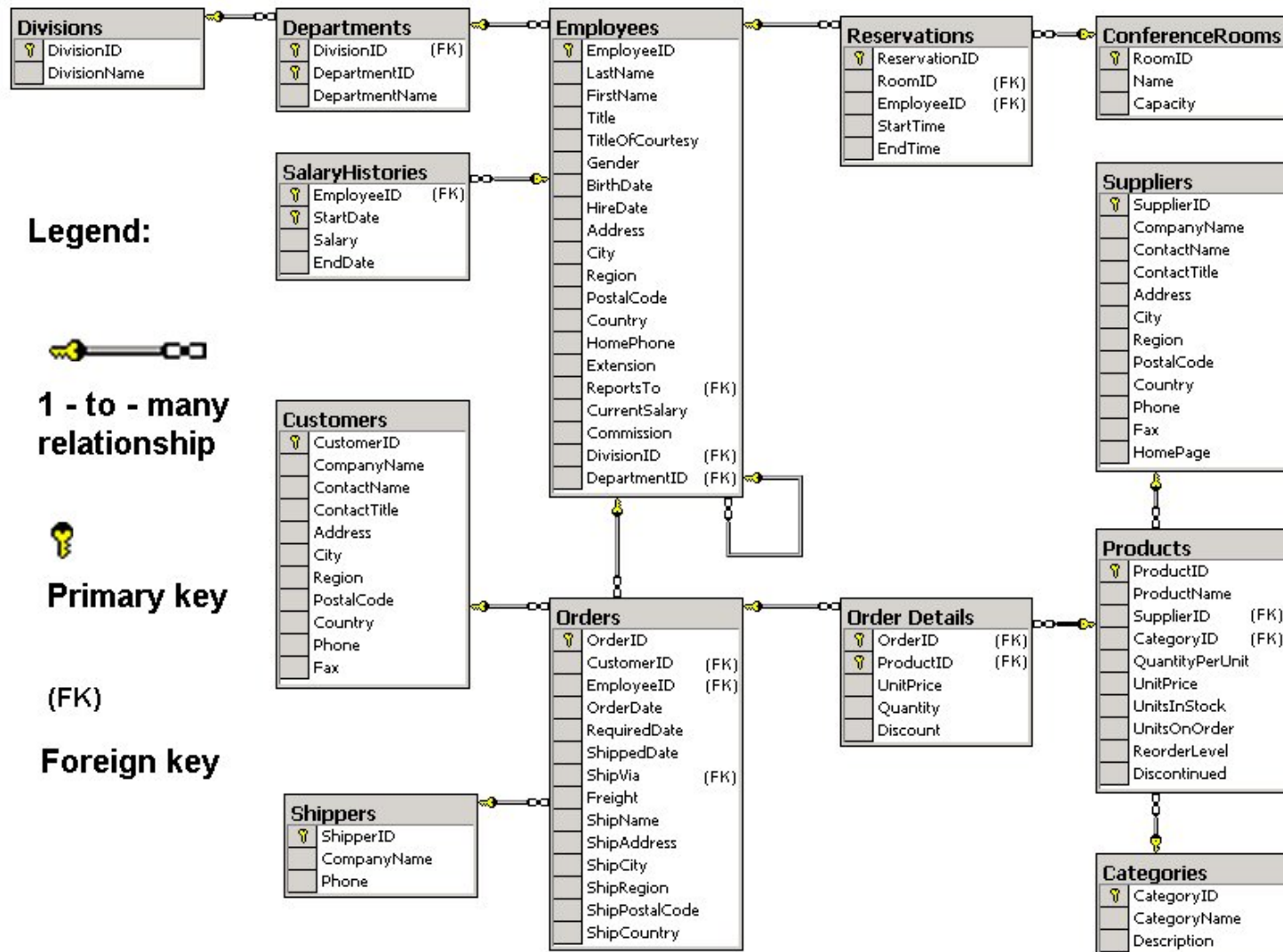
---

- ▶ **In this course, we will use a modified version of the Northwind database**
  - Northwind is an example database supplied with Microsoft Access and SQL Server
  - The database models a trading company that sells products to customers
  - Our version is called Eastwind and has been extended to provide data for our examples and exercises
- ▶ **Two of the tables, ConferenceRooms and Reservations, have not yet been created**
  - You will create and populate them in hands-on exercises

A blue pill-shaped button with a white border and a shadow, containing the text "Eastwind.sql".

Eastwind.sql

# The Course Database



# The Course Database

---

## ► Tables in the database

Table name	Primary key
Divisions	DivisionID
Departments	DivisionID+DepartmentID
Employees	EmployeeID
SalaryHistories	EmployeeID+StartDate
ConferenceRooms	RoomID
Reservations	ReservationID
Customers	CustomerID
Orders	OrderID
Shippers	ShipperID
Order Details	OrderID+ProductID
Products	ProductID
Suppliers	SupplierID
Categories	CategoryID

# The Course Database

## ► Relationships between tables in the database

Relationship	Foreign key
A Division can have many Departments Each Department must belong to one Division	DivisionID in Department references DivisionID in Division
A Department can have many Employees Each Employee must belong to one Department	DivisionID+DepartmentID in Employees references DivisionID+DepartmentID in Departments
An Employee has several Salary History entries Each Salary History entry belongs to one particular Employee	EmployeeID in SalaryHistories references EmployeeID in Employees
An Employee can have many Employees reporting to him or her An Employee can report to another Employee	ReportsTo in Employees references EmployeeID in Employees
A Conference Room can have many Reservations Each Reservation is for one particular Conference Room	RoomID in Reservations references RoomID in ConferenceRooms
An Employee can make several conference room Reservations Each Reservation is made by a particular Employee	EmployeeID in Reservations references EmployeeID in Employees

*(continued on the next slide)*



# The Course Database

---

## ► Relationships between tables in the database

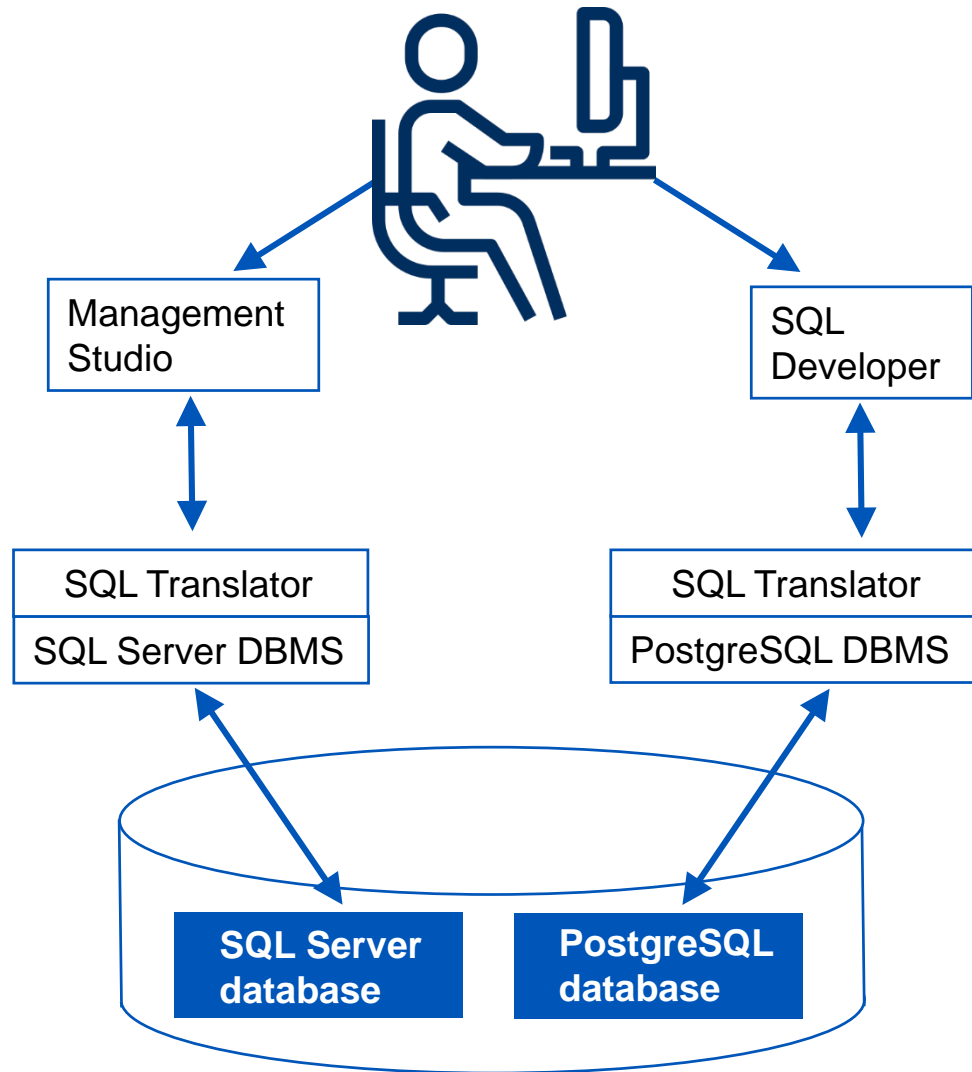
Relationship	Foreign key
A Customer can have many Orders Each Order belongs to one Customer	CustomerID in Orders references CustomerID in Customers
A Shipper can handle many Orders Each Order is handled by a particular Shipper	ShipVia in Orders references ShipperID in Shippers
An Employee can be responsible for many Orders Each Order has a particular Employee responsible for it	EmployeeID in Orders references EmployeeID in Employees
An Order can have several Order Detail items Each Order Detail item belongs to one particular Order	OrderID in Order Details references OrderID in Orders
A Product can occur in many Order Detail items Each Order Detail item is for a particular Product	ProductID in Order Details references ProductID in Products
A Category has many Products Each Product belongs to a Category	CategoryID in Products references CategoryID in Categories
A Supplier can supply many Products Each Product is being supplied by one particular Supplier	SupplierID in Products references SupplierID in Suppliers

# Technical Environment

---

- ▶ **The course computers are running the Windows Server 2012R2 operating system**
- ▶ **The installed database system is PostgreSQL**
  - The client tool for connecting to the database is SQL Developer
- ▶ **Microsoft SQL Server is also available for those who prefer to use that**
  - The client tool is Management Studio

# Technical Environment Illustration



DBMS = database management system

# Syntax Conventions

---

► **In this course, we will use the following conventions when describing SQL syntax**

- Keywords are spelled in uppercase
  - Example: `SELECT`
- Names to be substituted are spelled in lowercase and enclosed in brackets
  - Example: `<column>`
- Elements to be repeated a number of times are shown as commas with periods in between
  - Example: `<column>, ..., <column>`
- Optional elements are enclosed in square brackets
  - Example: `[DEFAULT <value>]`
- Either/or options are separated with a pipe sign
  - Example: `NULL | NOT NULL`

# Syntax Convention Example

- ▶ **The syntax for a simple SELECT statement can be described like this:**

```
SELECT * | <column>, ..., <column>  
FROM <table>  
[WHERE <condition>]
```

Columns separated by comma

- ▶ **The meaning of this is**

- The statement starts with the keyword SELECT, followed by either an asterisk or a comma-separated list of column names
- Next, the keyword FROM followed by a table name
- Finally, an optional clause starting with the keyword WHERE followed by a condition

Columns separated by comma

- ▶ **Example:**

```
SELECT FirstName, LastName  
FROM Employees  
WHERE CurrentSalary >= 6000;
```

c0-01.sql



# Comments

---

- ▶ **Comments in SQL are either enclosed between `/*` and `*/` or preceded by `--`**
  - `/* ... */` comments can span a number of lines or parts of a line of code
  - `--` comments go to the end of the line of code

- ▶ **Example**

c1-01.sql

```
SELECT /* This is a comment */ FirstName, LastName
/* This is a
multi-line comment */
FROM Employees -- This is a comment
-- This line is a comment
WHERE CurrentSalary >= 6000;
```

# Case Sensitivity

---

- ▶ **One of the more confusing areas of SQL is when to use uppercase or lowercase letters, and when it does not matter**
  - Here are the ground rules
- ▶ **SQL keywords**
  - SQL keywords are not case sensitive
  - Can be spelled in any combination of uppercase and lowercase letters
- ▶ **Examples:**
  - SELECT and FROM are keywords
  - The following will all work:  
  

```
SELECT * FROM Categories;  
select * from Categories;  
SelEct * froM Categories;
```
- ▶ **For clarity: In this course, keywords are consistently uppercase**

# Case Sensitivity

---

## ► PostgreSQL

- Table and column *names* are always lowercase
  - Stored as lowercase even if created as uppercase
  - Case insensitive when referred to after creation
- Table *contents* are always case sensitive

## ► Oracle

- Table and column *names* are always uppercase
  - Stored as uppercase even if created as lowercase
  - Case insensitive when referred to after creation
- Table *contents* are always case sensitive

## ► SQL Server

- Table and column names as well as table contents are always stored as they are entered
- SQL Server can be set up to be case sensitive or insensitive
  - Applies to both table and column *names*, as well as table *contents*

# Semicolon

---

- ▶ **The standard specifies that SQL statements should be terminated by a semicolon**
  - Unfortunately, the products do not follow the standard completely
- ▶ **SQL Server allows the semicolon, but does not require it**
- ▶ **In Oracle, the *client* tools interpret the semicolon as statement terminator, but the SQL translator in the DBMS does not recognize it**
  - Consequently, in SQLPlus, if you edit a statement and include a semicolon, you will get an error saying “invalid character” when you try to run the statement
  - The same happens in SQLPlus if you open a file with a semicolon using Open from the File menu, or using the GET command
    - The START command accepts the semicolon as statement terminator

# Display Cosmetics

---

- ▶ **When the client tools display the results of SQL queries, the cosmetic formatting may vary depending on the client tool and also on settings within each tool**
  - One example is the choice between grid and text formatting
- ▶ **Numeric values can vary considerably in the way decimals are displayed**
  - Some client tools will display trailing zero decimals; some will not
  - This may depend on data type and display settings
- ▶ **In real-life applications, SQL statements are embedded in host languages**
  - The cosmetic formatting will then be performed by the host language
  - The role of SQL is to extract the relevant data from the database, not to perform the formatting



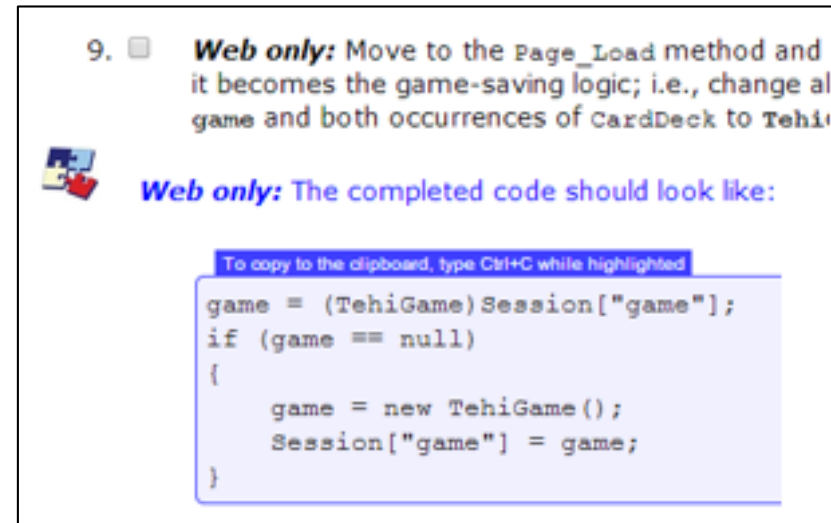
# AdaptaLearn™ Enabled

---

- ▶ **Electronic, interactive exercise manual**
- ▶ **Offers an enhanced learning experience**
  - Some courses provide folded steps that adapt to your skill level
  - Code is easily copied from the manual
  - After class, the manual can be accessed remotely for continued reference and practice
- ▶ **Printed and downloaded copies show all detail levels (hints and answers are unfolded)**



- ▶ **Launch AdaptaLearn by double-clicking its icon on the desktop**
  - Move the AdaptaLearn window to the side of your screen or shrink it to leave room for a work area for your development tools
- ▶ **Select an exercise from the exercise menu**
  - Zoom in and out of the AdaptaLearn window
  - Toggle between the AdaptaLearn window and your other windows
- ▶ **Look for a folded area introduced with blue text (not available in all courses)**
  - Click the text to see how folds work
- ▶ **Try to copy and paste text from the manual**
  - Some courses have code boxes that make it easy to copy areas of text while highlighted (as shown)



## Hands-On Exercise 1.1

**In your Exercise Manual, please refer to Hands-On Exercise 1.1: Tools and Environment**

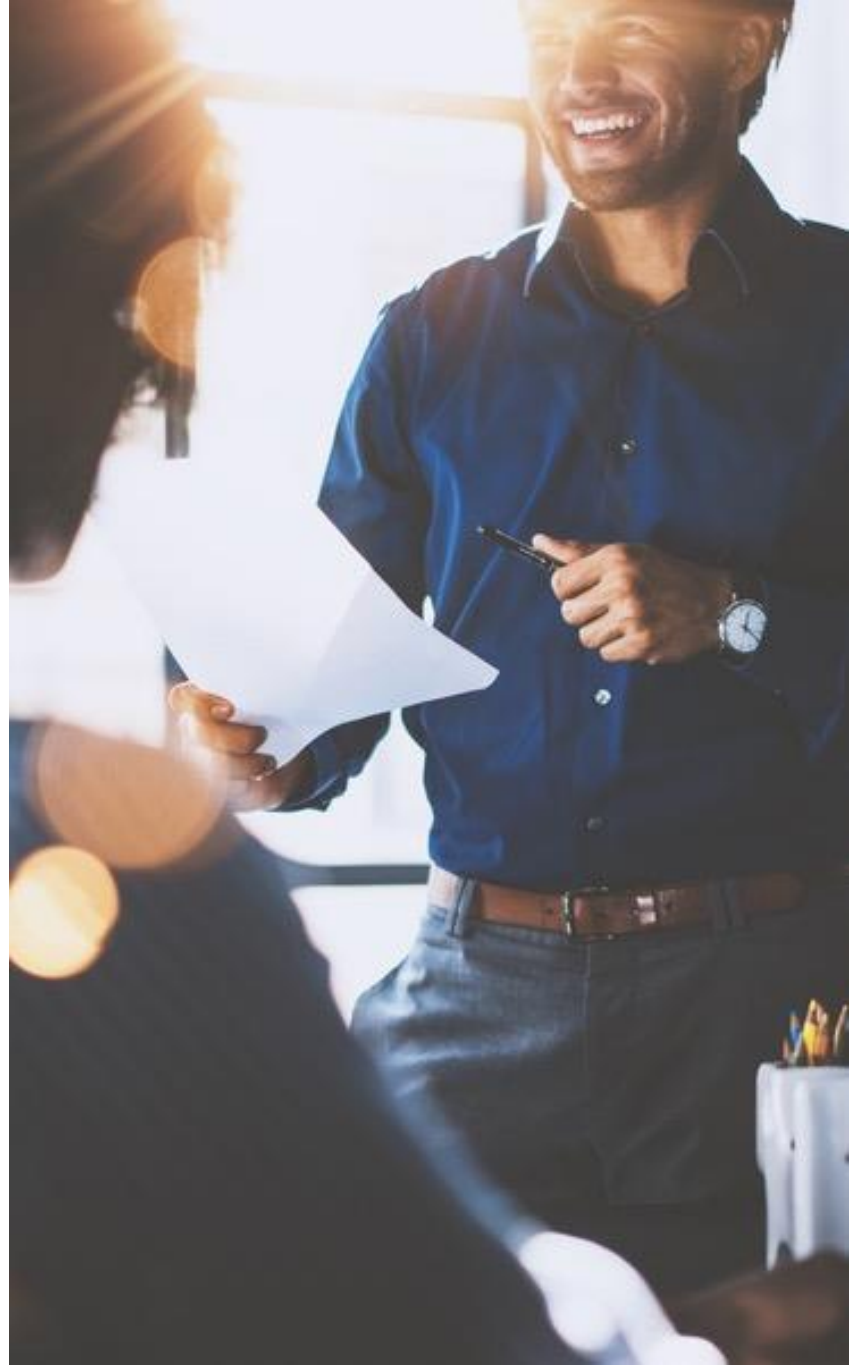
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ **Describe why SQL is important**
- ▶ **Review the fundamentals of database theory**
- ▶ **Review the evolution and history of SQL**
- ▶ **Describe the course environment**





## Chapter 2

# Data Definition



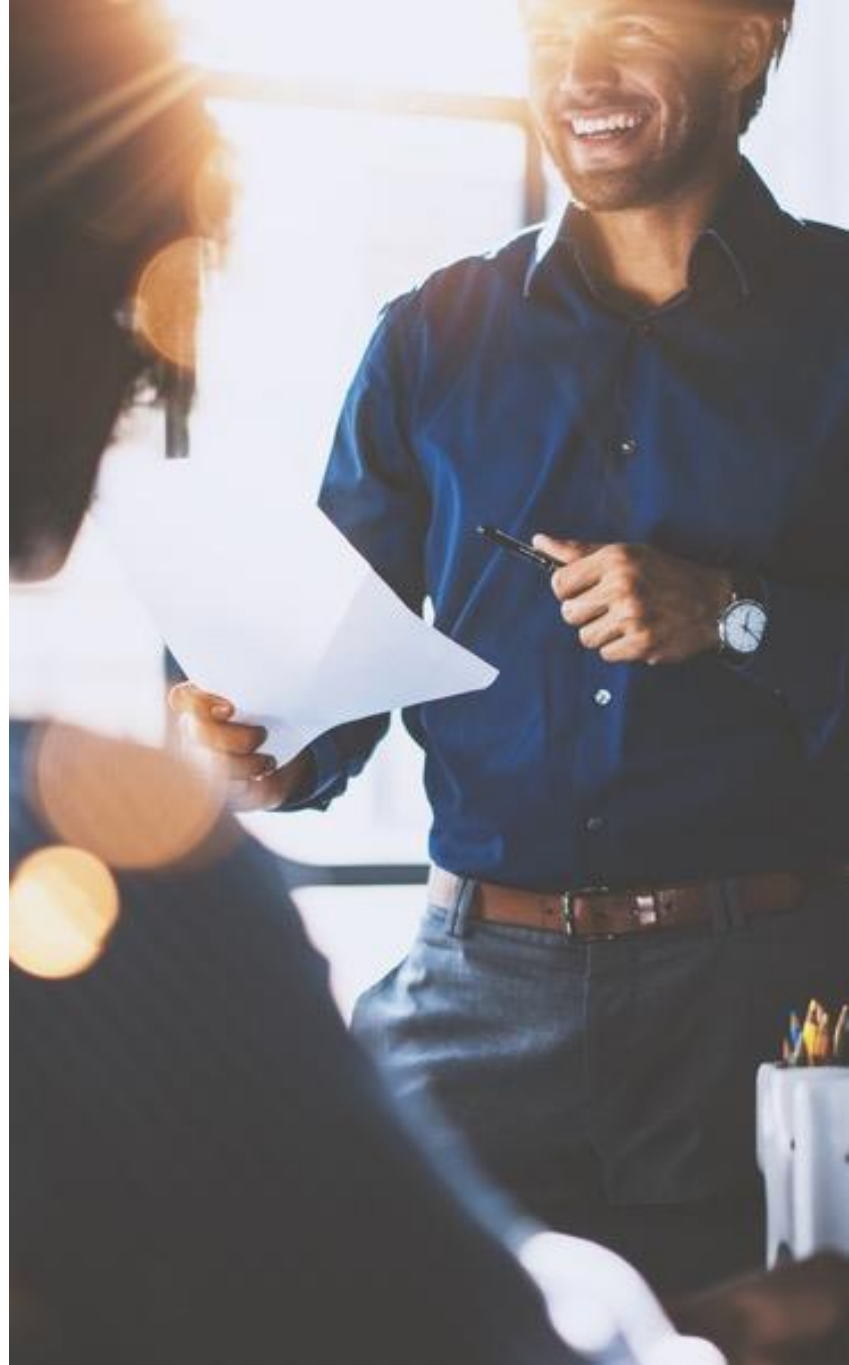
LEARNING TREE<sup>™</sup>  
INTERNATIONAL



# Objectives

---

- ▶ Learn about column data types
- ▶ Create tables and columns
- ▶ Modify and remove tables
- ▶ Add constraints to tables
- ▶ Create indexes



# Contents

---

## Data Types

- ▶ Creating Tables
- ▶ Modifying and Dropping Tables
- ▶ Constraints
- ▶ Indexes



# Data Types

---

- ▶ **Each database column must have a valid data type**
  - Data types fall into categories
    - Character
    - Numeric
    - Date and time
    - Binary
    - Specialized
- ▶ **Data types are only partly standardized**
  - Each product and product version has differences
  - Similar data type names can have slightly different meanings in different products
- ▶ **The list on the next slide is meant as an overview only**
  - Product specifics like sizes, limits, and implementation details are not included
  - Some specialized data types are not included
  - Check the product documentation for details

# ANSI/ISO Standard Data Types

- ▶ **Some numeric data types allow specifying precision and scale (p, s)**
  - Precision is the number of significant digits
  - Scale is the number of decimals

Numeric	Character	Date and time	Binary
INT SMALLINT NUMERIC(p, s) DECIMAL(p, s) FLOAT(p) REAL DOUBLE	CHAR(length) VARCHAR(length) NCHAR(length) NVARCHAR(length)	DATE TIME TIMESTAMP	BIT BIT VARYING

# Contents

---

- ▶ Data Types

## Creating Tables

- ▶ Modifying and Dropping Tables
- ▶ Constraints
- ▶ Indexes



# CREATE TABLE Statement

- ▶ **The CREATE TABLE statement defines the structure of a table**
  - Updates the system catalog, also known as the data dictionary

- ▶ **Syntax:**

```
CREATE TABLE <table name>
```

```
(<column name> <data type> [DEFAULT <value>] [NULL | NOT NULL]  
, ...  
, <column name> <data type> [DEFAULT <value>] [NULL | NOT NULL]);
```

Columns separated by comma

Column list enclosed  
in parentheses

- ▶ **The CREATE TABLE statement fails if the table already exists**



# CREATE TABLE Statement

---

- ▶ **The column list must be enclosed in parentheses, even if there is only one column**
- ▶ **Each column must have a valid name and data type**
  - Depending on the data type, size may or may not be specified
- ▶ **A column can have a default value**
  - Whenever a new row is inserted, unspecified columns will be set to the default value
  - If no default value is defined for the column, NULL is used
- ▶ **A column can be mandatory (NOT NULL)**

# CREATE TABLE Examples

## ► Examples

c2-01.sql

```
CREATE TABLE Divisions
```

```
(DivisionID      NUMERIC(5)  NOT NULL  
,DivisionName   VARCHAR(40) NOT NULL);
```

```
CREATE TABLE Departments
```

```
(DivisionID      NUMERIC(5)  NOT NULL  
,DepartmentID   NUMERIC(5)  NOT NULL  
,DepartmentName VARCHAR(40) NOT NULL);
```

DivisionID	DivisionName

DivisionID	DepartmentID	DepartmentName

# CREATE TABLE Example

```
CREATE TABLE Employees
(EmployeeID      INT          NOT NULL
,LastName        VARCHAR(20)  NOT NULL
,FirstName       VARCHAR(10)  NOT NULL
,Title           VARCHAR(25)
,TitleOfCourtesy VARCHAR(5)
,Gender          CHAR(1)
,BirthDate       TIMESTAMP
,HireDate        TIMESTAMP
,Address         VARCHAR(30)
,City            VARCHAR(15)
,Region          VARCHAR(10)
,PostalCode      VARCHAR(10)
,Country         VARCHAR(15)
,HomePhone       VARCHAR(20)
,Extension       VARCHAR(4)
,ReportsTo       INT
,CurrentSalary   NUMERIC(5)   NOT
NULL
,Commission      NUMERIC(5)
,DivisionID      NUMERIC(5)
,DepartmentID    NUMERIC(5));
```

## ► Example

c2-02.sql

- TIMESTAMP is standard
- In SQL Server, DATETIME is similar

# CREATE TABLE as SELECT

c2-03.sql

► **We have seen that the result of a SELECT statement is in table format**

- A table can be created directly based on a SELECT result
- Column names and data types are derived from the SELECT statement
- The rows in the result are copied into the new table and stored

► **Standard syntax**

```
CREATE TABLE <table name> AS  
SELECT ... FROM ...
```

► **Example**

```
CREATE TABLE HighPaid AS  
SELECT FirstName, LastName  
FROM Employees  
WHERE CurrentSalary >= 6000;
```

► **SQL Server syntax**

```
SELECT ...  
INTO <table name>  
FROM ...
```

► **Example**

```
SELECT FirstName, LastName  
INTO HighPaid  
FROM Employees  
WHERE CurrentSalary >= 6000;
```

# Contents

---

- ▶ Data Types
- ▶ Creating Tables

## Modifying and Dropping Tables

- ▶ Constraints
- ▶ Indexes



# ALTER TABLE Statement

---

- ▶ **The ALTER TABLE statement modifies the structure of a table**
  - Updates the system catalog
- ▶ **Possible modifications are**
  - Adding or removing columns
  - Increasing or decreasing column size
  - Changing column data type
  - Adding or removing constraints
    - Including NOT NULL
- ▶ **Product- and version-specific restrictions apply to**
  - Removing columns
  - Decreasing column size
  - Changing column data type



# Adding Columns

---

## ► Syntax

```
ALTER TABLE <table name>
```

```
ADD
```

```
<column name> <data type> [DEFAULT <value>] [NULL | NOT NULL];
```

- A new column without a DEFAULT value will contain NULL value
  - The new column will not be populated, so only the system catalog is updated
- NOT NULL is allowed only if there is also a DEFAULT value
  - The new column will then be populated with the default value

## ► Example

```
ALTER TABLE Divisions ADD Description VARCHAR(20);
```



c2-04.sql

# Modifying Columns

---

## ► Syntax

```
ALTER TABLE <table name>  
ALTER COLUMN  
<column name> <data type> [DEFAULT <value>] [NULL | NOT NULL];
```

## ► PostgreSQL requires the TYPE keyword

## ► Oracle deviates from the standard and uses the keyword MODIFY instead of ALTER COLUMN

## ► Example

c2-05.sql

- PostgreSQL

```
ALTER TABLE Divisions ALTER COLUMN Description TYPE VARCHAR(80);
```

- SQL Server

```
ALTER TABLE Divisions ALTER COLUMN Description VARCHAR(80);
```

- Oracle

```
ALTER TABLE Divisions MODIFY Description VARCHAR(80);
```

# Removing Columns

---

▶ A column can be removed from the table

▶ **Syntax**

```
ALTER TABLE <table name> DROP COLUMN <column name>;
```

▶ **Example**

```
ALTER TABLE Divisions DROP COLUMN Description;
```

c2-06.sql

# DROP TABLE Statement

---

- ▶ **Removes the table definition as well as table content**
  - Updates the system catalog
- ▶ **Syntax**  
`DROP TABLE <table name>;`
- ▶ **Example**  
`DROP TABLE Divisions;`

c2-07.sql

# Checking Table Definitions

---

- ▶ **Table definitions are stored in the system catalog (data dictionary)**
  - Can be queried using SQL statements
  - Very different in different products
- ▶ **ANSI/ISO standard INFORMATION\_SCHEMA**  
`SELECT ... FROM INFORMATION_SCHEMA.<view>;`
  - Not supported by Oracle
- ▶ **Product-dependent tools for checking table definitions**
  - SQL Server  
`SP_HELP <table name>;`
  - Oracle SQL\*Plus  
`DESCRIBE <table name>`
- ▶ **Most products have GUI tools that can easily display system catalog information**
  - Many third-party tools exist

## Hands-On Exercise 2.1

**In your Exercise Manual, please refer to Hands-On Exercise 2.1: Creating and Altering Tables**

- ▶ **Solutions to all the exercises are provided as files located in the folder C:\Course925Solutions**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**





# Contents

---

- ▶ Data Types
- ▶ Creating Tables
- ▶ Modifying and Dropping Tables

## Constraints

- ▶ Indexes



# Constraints

---

- ▶ **Constraints are used to enforce valid data in columns**
- ▶ **Constraint definition is specified in the 1989 addendum to the ANSI/ISO SQL standard**
  - Was added to products after the standard was established
- ▶ **Different types of constraints**
  - NOT NULL
  - CHECK
  - PRIMARY KEY
  - UNIQUE
  - FOREIGN KEY
- ▶ **NOT NULL is actually a special type of check constraint, but is specified slightly different for historical reasons**
  - Was available in most products already before the standard

# Constraint Definition

---

- ▶ **Constraints can be specified in the CREATE TABLE statement**
  - Two alternative ways
    - As part of the column specification
      - NOT NULL constraints are usually specified this way
    - As a separate specification
      - Useful for keys that involve more than one column
- ▶ **Constraints can be specified with an ALTER TABLE ... ADD statement**
  - The preferred way for most constraint types
  - Provides flexibility in the definition

# NOT NULL Constraint

---

- ▶ **Usually specified directly with the column in CREATE TABLE**
  - As in the previous example of CREATE TABLE
- ▶ **Specifies that the column does not allow NULL values**
  - Particularly relevant for primary key
    - The *entity integrity* rule specifies that a primary key may not have NULL values
  - Relevant for foreign key if the relationship is mandatory
  - Other columns may also be mandatory
- ▶ **NULL means *unknown* or *missing* value**
  - Not the same as blank or zero



# CHECK Constraint

## ▶ Enforce business rules by preventing illegal column values

- Column value can be checked against constants
- Column value can be checked against other column values within the same row
- Cannot check against other rows or other tables

## ▶ It is a good practice to give meaningful names to constraints

## ▶ Syntax

```
ALTER TABLE <table name>  
ADD CONSTRAINT <constraint name>  
CHECK (<condition>);
```

## ▶ Example

```
ALTER TABLE Employees  
ADD CONSTRAINT CK_Salary_Commission  
CHECK (Commission <= CurrentSalary);
```

This prefix is just a naming convention

c2-08.sql

# PRIMARY KEY and UNIQUE Constraints

---

## ► PRIMARY KEY constraint

- Enforces uniqueness of primary key
- One column or a combination of columns
- A table can have only one primary key constraint
  - But it can include more than one column

## ► UNIQUE constraint

- Enforces uniqueness of alternate keys
- In addition to primary key
- One column or a combination of columns
- A table can have several unique constraints in addition to the primary key constraint

## ► Both PRIMARY KEY and UNIQUE constraints implicitly build a unique index

- The only practical way of enforcing uniqueness



# PRIMARY KEY and UNIQUE Constraints

## ► Syntax

```
ALTER TABLE <table name>  
ADD CONSTRAINT <constraint name>  
PRIMARY KEY | UNIQUE (<column>, ..., <column>);
```

## ► Examples

```
ALTER TABLE Divisions  
ADD CONSTRAINT PK_Divisions  
PRIMARY KEY (DivisionID);
```

This prefix is just a  
naming convention

```
ALTER TABLE Divisions  
ADD CONSTRAINT UK_DivisionName  
UNIQUE (DivisionName);
```

This prefix is just a  
naming convention

c2-09.sql

# FOREIGN KEY Constraint

- ▶ Specifies a relationship between tables
- ▶ A foreign key references the *primary key* of the related table
  - One column or a combination of columns

- ▶ **Syntax**

```
ALTER TABLE <table name>  
ADD CONSTRAINT <constraint name>  
FOREIGN KEY (<column>,...<column>)  
REFERENCES <table> (<column>,...,<column>);
```

List of referenced columns is not required if it is the primary key

- ▶ **Example**

```
ALTER TABLE Products  
ADD CONSTRAINT FK_Product_Category  
FOREIGN KEY (CategoryID)  
REFERENCES Categories;
```

c2-10.sql

# Referential Integrity

---

- ▶ **Foreign key constraints enforce the *referential integrity* rule of relationships**
  - A foreign key may not reference a nonexistent value
- ▶ **In order to enforce referential integrity, the following checks are required**
  - Whenever a foreign key is inserted or updated
    - Check that the matching primary key exists
  - Whenever a primary key is deleted or updated
    - Options specify what happens if there are foreign key values that reference it
    - Various degrees of support in different products and versions



# Contents

---

- ▶ Data Types
- ▶ Creating Tables
- ▶ Modifying and Dropping Tables
- ▶ Constraints

## Indexes



# Indexes

---

- ▶ **In general, the rows in a table are unsorted**
  - Finding a particular value requires searching the complete table
  - Indexes can be created to speed up the search
- ▶ **An index is sorted on the indexed value and has pointers to the rows containing that value (similar to an index in a book)**
- ▶ **Index contents are automatically maintained when the table is updated**
- ▶ **Alternatively, you may store the entire table rows inside the index structure**
  - Oracle: index-organized table
  - SQL Server: clustered index

- ▶ **Indexes can be created on a single column or a combination of columns**
  - A table can have several different indexes on different columns
- ▶ **Indexes can be unique or non-unique**
  - PRIMARY KEY and UNIQUE constraints automatically create unique indexes
  - Non-unique indexes have no logical meaning and only affect performance
    - Can be created or dropped without affecting application logic
- ▶ **Index usage is transparent to the SQL statements**
  - The SQL translator optimizes each SQL statement to take advantage of existing indexes
  - No extra code is required for indexes to be used
  - If indexes are created or dropped, the optimizer automatically adapts



# Creating and Dropping Indexes

## ► CREATE INDEX syntax

```
CREATE INDEX <index name> ON <table> (<column>, ..., <column>);
```

## ► Examples

c2-11.sql

```
CREATE INDEX Ind_ProductName ON Products(ProductName);
```

```
CREATE INDEX Ind_EmployeeName ON Employees(LastName, FirstName);
```

## ► Index names

- In PostgreSQL and Oracle, index names are unique within the database schema
- In SQL Server, index names are unique within each table
- Therefore, the DROP syntax is slightly different

## ► DROP INDEX syntax

- PostgreSQL and Oracle

```
DROP INDEX <index name>;
```

- SQL Server

```
DROP INDEX <table name>.<index name>;
```

# Index Advantages and Disadvantages

---

## ► Advantages

- Speed up search for a particular value
- Improvement increases with selectivity
- Can also speed up retrieval in sorted order

## ► Disadvantages

- Slow down inserts and updates
- Take up storage space

## Hands-On Exercise 2.2

**In your Exercise Manual, please refer to Hands-On Exercise 2.2: Constraints and Indexes**

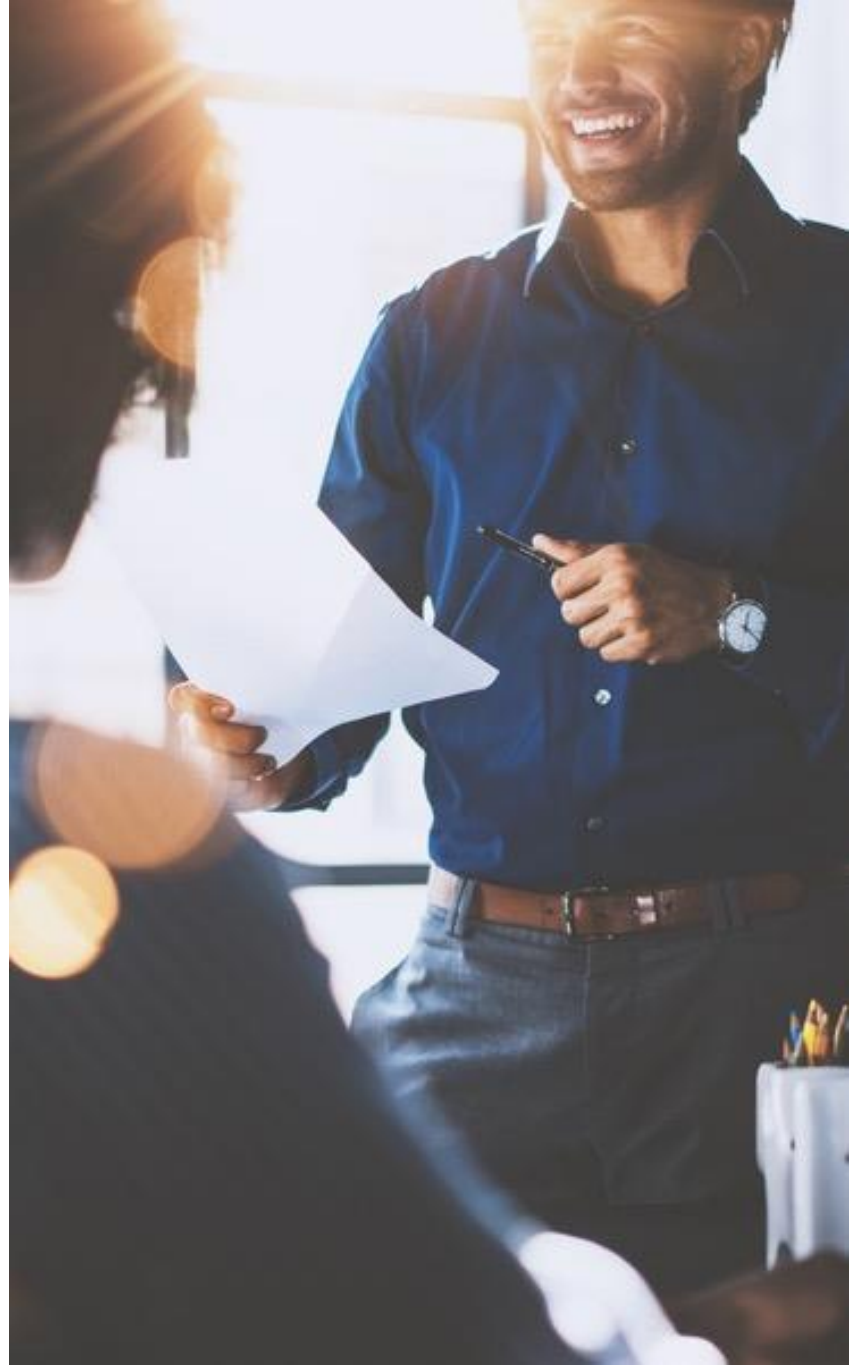
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Learn about column data types
- ▶ Create tables and columns
- ▶ Modify and remove tables
- ▶ Add constraints to tables
- ▶ Create indexes



# Chapter 3

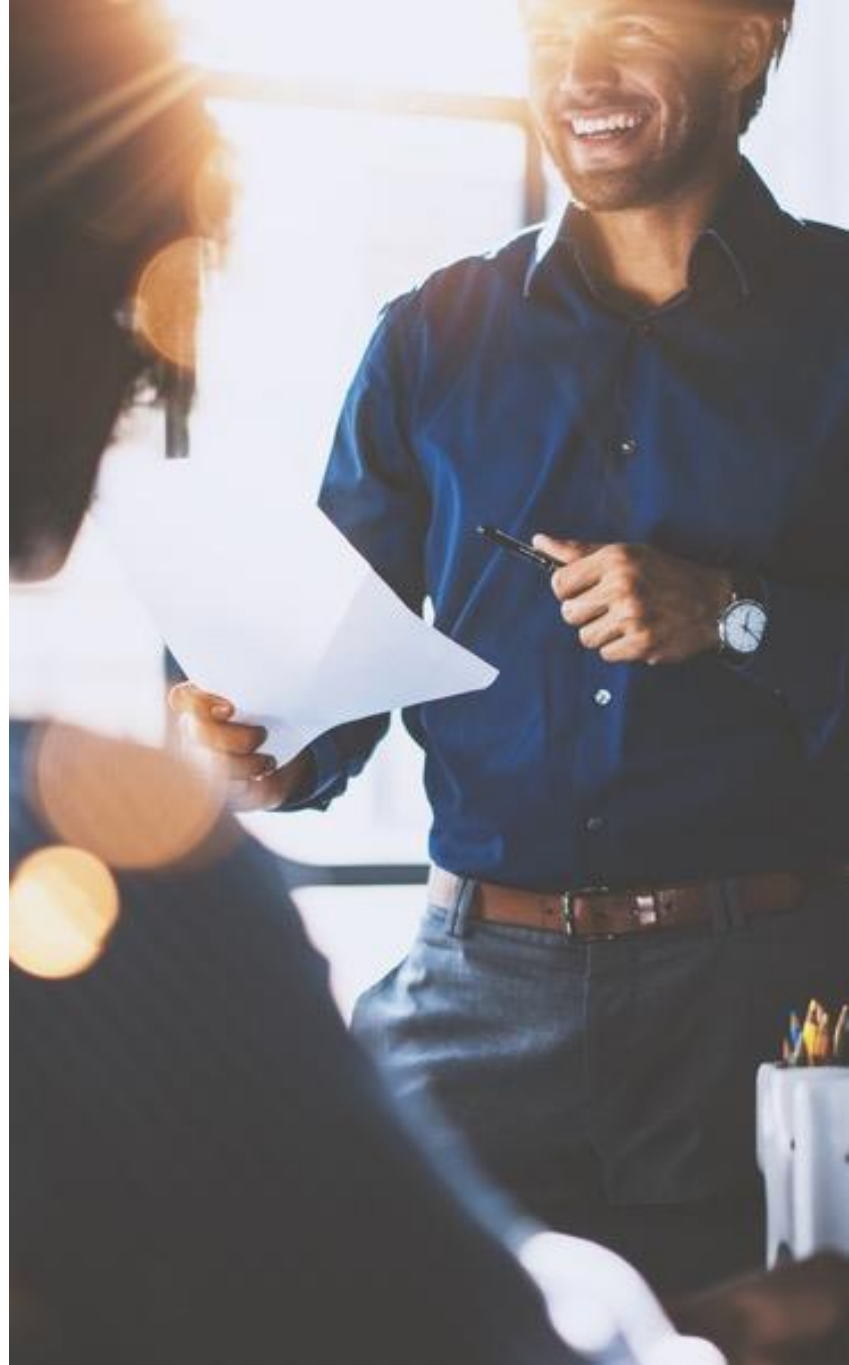
## Data Modification



# Objectives

---

- ▶ **Use INSERT to add new rows**
- ▶ **Use UPDATE to modify rows**
- ▶ **Use DELETE to remove rows**
- ▶ **Learn about transaction handling**





# Contents

---

## INSERT, UPDATE, DELETE

### ► Transactions



# Data Modification Statements

---

- ▶ **Three different statements modify the contents of a table**
  - INSERT: Adds new rows
  - UPDATE: Modifies values in existing rows
  - DELETE: Removes rows

# INSERT Statement

---

- ▶ **The INSERT statement adds new rows to a table**
- ▶ **It comes in two flavors**
  - Using VALUES to add exactly one row with specified values
  - Using SELECT to add a number of rows from the result of a query
- ▶ **In both flavors, list of column names is optional**
  - If omitted, assumes all columns in the table
    - In the same column order as when the table was created
  - Recommended to always specify columns
    - Columns may be omitted
    - Column order is not dependent on table
    - Will still work if new columns are added to the table
- ▶ **In both flavors, the number of column values must match the number of columns**

# INSERT Statement

---

## ► Syntax

```
INSERT INTO <table> [(<column>, ..., <column>)]  
VALUES (<value>, ..., <value>);
```

```
INSERT INTO <table> [(<column>, ..., <column>)]  
<SELECT statement>;
```

## ► Examples

```
INSERT INTO Departments  
(DivisionID, DepartmentID, DepartmentName)  
VALUES(1, 100, 'Accounting');
```

```
INSERT INTO Highpaid  
SELECT FirstName, LastName  
FROM Employees  
WHERE CurrentSalary >= 6000;
```

c3-01.sql

# UPDATE Statement

## ► The UPDATE statement modifies the column values in existing rows

- Uses a WHERE condition to determine which rows to update
- Most commonly WHERE <primary key> = <value>
- Without a WHERE clause, *all* the rows in the table will be updated

## ► Syntax

```
UPDATE <table>  
SET <column> = <value>, ..., <column> = <value>  
WHERE <condition>;
```

Columns separated by comma

c3-02.sql

## ► Example

```
UPDATE Employees  
SET    CurrentSalary = CurrentSalary + 100  
      , TitleOfCourtesy = 'Ms.'  
WHERE  EmployeeID = 4;
```

Specifies which row(s) to update

Columns separated by comma

# DELETE Statement

---

► **The DELETE statement removes rows from a table**

- Uses a WHERE condition to determine which rows to delete
- Most commonly WHERE <primary key> = <value>
- Without a WHERE clause, *all* the rows in the table will be deleted

► **Syntax**

```
DELETE FROM <table>  
WHERE <condition>;
```

► **Example**

```
DELETE FROM Employees  
WHERE EmployeeID = 7;
```

Specifies which row(s) to delete

c3-03.sql



# Contents

---

► INSERT, UPDATE, DELETE

## Transactions



# Transactions

---

- ▶ **A *transaction* is a set of INSERT/UPDATE/DELETE operations that belong together as a logical *unit* of work**
  - All or nothing
- ▶ **Commands for handling transactions**
  - COMMIT [WORK]; ends the transaction with success and makes the updates permanent
  - ROLLBACK [WORK]; ends the transaction with failure and undoes updates already made
  - The keyword WORK is not required, just using COMMIT and ROLLBACK is sufficient

# Product Specifics

---

## ► Standard

- A transaction starts implicitly with the first update after a previous COMMIT/ROLLBACK
- Oracle follows the standard

## ► SQL Server

- By default, each statement is a transaction in itself and is automatically committed if succeeding
- If a transaction should encompass several statements, then the start of the transaction is specified using the command, BEGIN TRANSACTION

## ► PostgreSQL

- By default, each statement is a transaction in itself and is automatically committed if succeeding
- If a transaction should encompass several statements, then the start of the transaction is specified using the command, BEGIN TRANSACTION
- BEGIN TRANSACTION is actually a procedural command, not an SQL statement

# Handling Transactions in Programs

---

- ▶ **Whenever a program contains several updates as part of a transaction, the program should**
  - Start the transaction
    - In SQL Server, use `BEGIN TRANSACTION`
  - Check the return status of each update
  - If failure, do `ROLLBACK` and abort the processing
  - If success, continue
  - When all parts of the transaction have completed with success, do `COMMIT`
  
- ▶ **The database automatically performs a rollback if some serious failure occurs during a transaction**
  - Server failure
  - Client failure
  - Network failure
  - Program failure

## Hands-On Exercise 3.1

**In your Exercise Manual, please refer to Hands-On Exercise 3.1: Updating Table Contents**

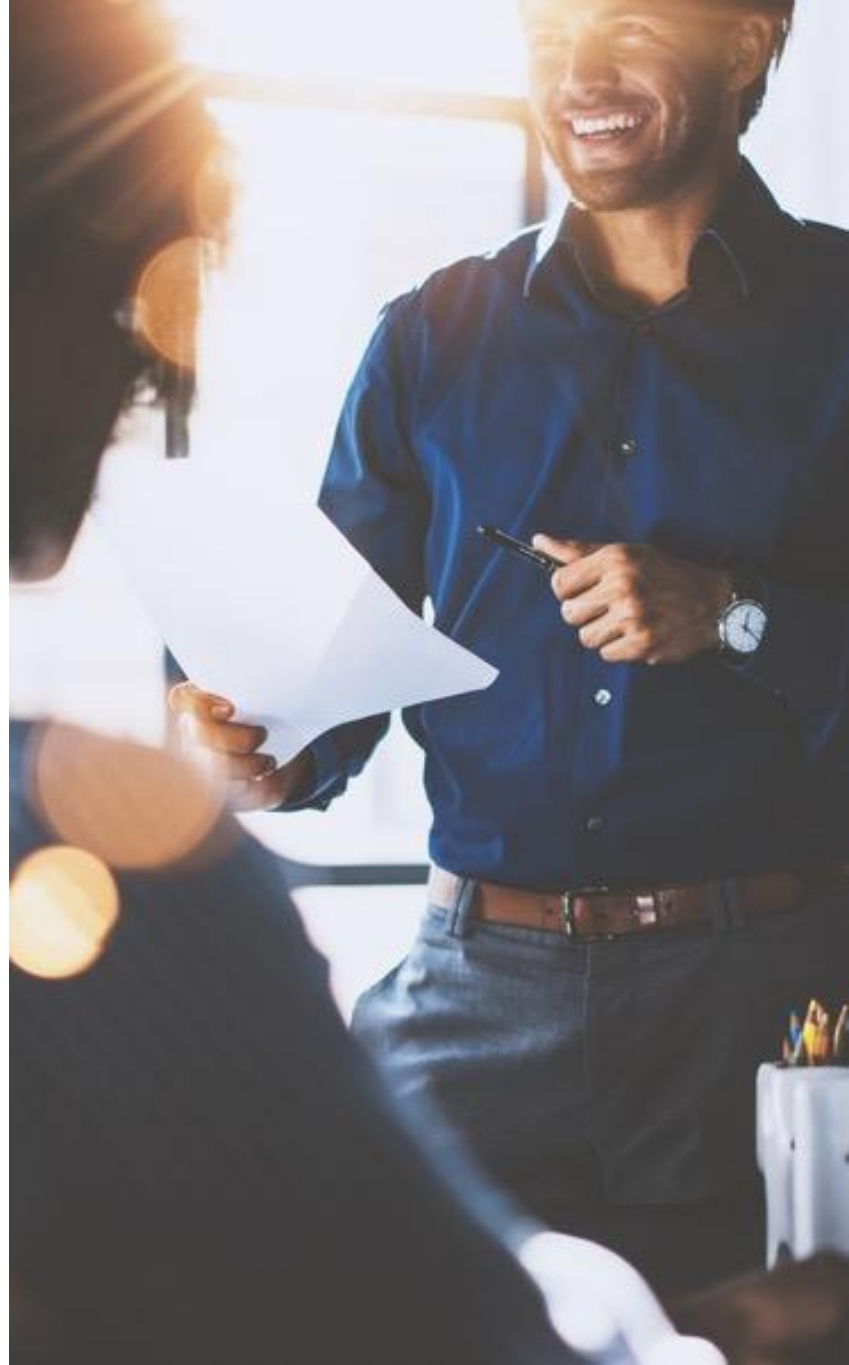
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Use INSERT to add new rows
- ▶ Use UPDATE to modify rows
- ▶ Use DELETE to remove rows
- ▶ Learn about transaction handling





# Chapter 4

## Single-Table SELECT Statements



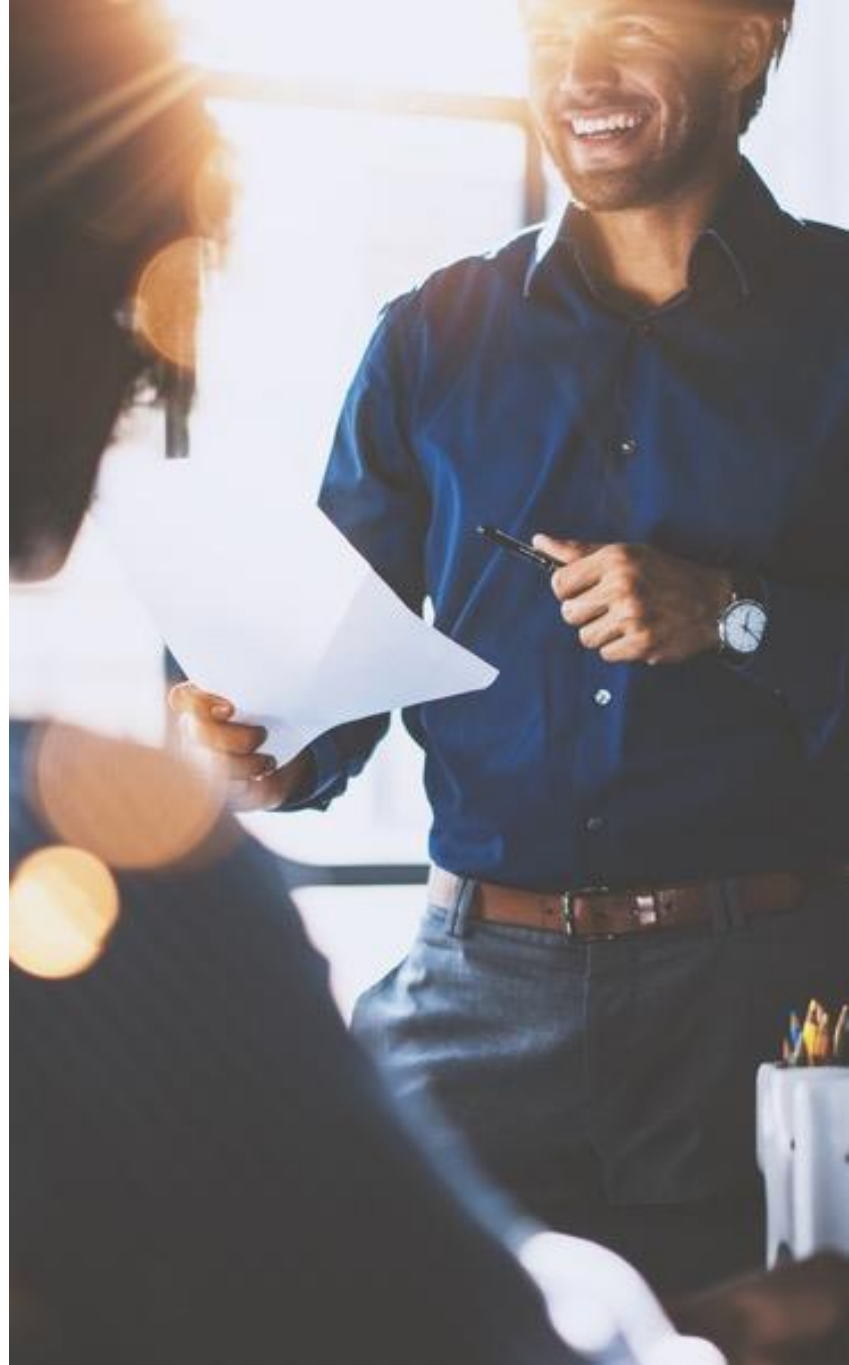
LEARNING TREE™  
INTERNATIONAL



# Objectives

---

- ▶ Describe the basic structure of a **SELECT** statement
- ▶ Retrieve columns and expressions
- ▶ Apply filter conditions
- ▶ Sort the result set
- ▶ Describe issues with **NULL** values
- ▶ Use **DISTINCT** to suppress duplicate rows



# Contents

---

## SELECT Statement Structure

- ▶ Columns and Expressions
- ▶ WHERE Filter Conditions
- ▶ ORDER BY to Specify Sorting
- ▶ NULL Values
- ▶ DISTINCT



# SELECT Statement

## ► The SELECT statement retrieves data from database tables

- What it actually does is define a result set with table structure and contents

## ► Example

c0-01.sql

- If we have this table and this SELECT statement  
`SELECT FirstName, LastName  
FROM Employees  
WHERE CurrentSalary >= 6000;`

EmployeeID	LastName	FirstName	CurrentSalary
1	Davolio	Nancy	2000
2	Fuller	Andrew	9000
3	Leverling	Janet	6000
4	Peacock	Margaret	3000
5	Buchanan	Steven	5000
6	Suyama	Michael	3600
7	King	Robert	3000
8	Callahan	Laura	4000
9	Dodsworth	Anne	5200

- Will produce this result

FirstName	LastName
Andrew	Fuller
Janet	Leverling

# Overview of the SELECT Statement

---

- ▶ **The SELECT clause is always required, with at least one column expression**
  - The other clauses are used only when needed
  - In Oracle, the FROM clause is also required
- ▶ **The clauses must appear in the order shown below**

**SELECT**    <column expression>,...,<column expression> | \*

**FROM**      <table> [JOIN <table> ON <join condition>]

**WHERE**     <filter condition>

**GROUP BY** <column>,...,<column>

**HAVING**    <filter condition>

**ORDER BY** <column>,...,<column>

# SELECT Examples

c4-01.sql

SELECT \*  
FROM Divisions;

\* means all the  
table's columns

DivisionID	DivisionName
1	America
2	Europe
3	Pacific

SELECT DivisionName  
,DivisionID  
FROM Divisions;

DivisionName	DivisionID
America	1
Europe	2
Pacific	3

Columns separated  
by comma

# Contents

---

- ▶ **SELECT Statement Structure**

## **Columns and Expressions**

- ▶ **WHERE Filter Conditions**
- ▶ **ORDER BY to Specify Sorting**
- ▶ **NULL Values**
- ▶ **DISTINCT**





# Column Expression

---

- ▶ **SELECT \*** will return all the columns in the table
- ▶ **More common to specify column expressions**
  - Include the required columns only
  - This is called a *projection*
- ▶ **A column expression can be**
  - A plain column
  - A constant
  - A subquery
  - A combination of the above



# Column Alias

---

- ▶ **A column expression can have an alias name**
  - `SELECT <column name> AS <alias name>, ...`
  - The alias name will become the column name in the result set
  - The keyword `AS` is specified in the standard, but most products do not require it
  
- ▶ **Alias names are particularly useful for complex expressions**
  
- ▶ **If an expression is not given an alias name**
  - PostgreSQL returns `?column?` as the column name
  - SQL Server returns a blank column name
  - Oracle returns the expression as the column name



Column alias names serve to name the column in the result



It is not permitted to reference the column alias from inside the statement  
except in `ORDER BY`

---

# Column Expression and Alias Example

```
SELECT EmployeeID
       ,FirstName
       ,LastName
       ,CurrentSalary*12 AS YearlySalary
FROM   Employees;
```

c4-02.sql

EmployeeID	FirstName	LastName	YearlySalary
1	Nancy	Davolio	24000.0000
2	Andrew	Fuller	108000.0000
3	Janet	Leverling	72000.0000
4	Margaret	Peacock	36000.0000
5	Steven	Buchanan	60000.0000
6	Michael	Suyama	43200.0000
7	Robert	King	36000.0000
8	Laura	Callahan	48000.0000
9	Anne	Dodsworth	62400.0000

# Accidental Alias

## ► Column expressions are separated by commas

c4-03.sql

- If commas are omitted, it may not necessarily give a syntax error
- Since the keyword AS is not required, a column name could be taken as an alias name

## ► Example

```
SELECT DivisionID  
       ,DivisionName  
FROM   Divisions;
```

DivisionID	DivisionName
1	America
2	Europe
3	Pacific

No comma

```
SELECT DivisionID  
       DivisionName  
FROM   Divisions;
```

DivisionName
1
2
3

This is now an alias

# String Concatenation

- ▶ **The standard operator for string concatenation is the double pipe (||)**
  - SQL Server deviates from the standard and uses a plus sign as concatenation operator

- ▶ **Example**

c4-04.sql

```
SELECT FirstName || ' ' || LastName AS EmployeeName  
FROM Employees;
```

SQL Server uses +  
instead of ||

```
EmployeeName  
-----  
Andrew Fuller  
Steven Buchanan  
Nancy Davolio  
Janet Leverling  
Margaret Peacock  
Michael Suyama  
Robert King  
Laura Callahan  
Anne Dodsworth
```

## Hands-On Exercise 4.1

**In your Exercise Manual, please refer to Hands-On Exercise 4.1: Columns and Expressions**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Contents

---

- ▶ **SELECT Statement Structure**
- ▶ **Columns and Expressions**

## **WHERE Filter Conditions**

- ▶ **ORDER BY to Specify Sorting**
- ▶ **NULL Values**
- ▶ **DISTINCT**



# The WHERE Clause

## ► The WHERE clause specifies a filter condition

- Also called *restriction* or *selection*
- The result set will include only rows that satisfy the condition

## ► Test for equality on the primary key is a commonly used condition

- All sorts of conditions are possible

## ► Example

- Retrieve information about employee 3

```
SELECT EmployeeID, FirstName, LastName
       ,CurrentSalary*12 AS YearlySalary
FROM   Employees
WHERE  EmployeeID = 3;
```

EmployeeID	FirstName	LastName	YearlySalary
3	Janet	Leverling	72000.0000



Column alias names may not be referenced in WHERE conditions

c4-05.sql



# Conditional Operators

---

## ► Operators used in the WHERE clause

- Test for equality, inequality, greater or less: = <> > < >= <=
- Test for several values: IN and NOT IN
- Intervals: BETWEEN and NOT BETWEEN
- NULL values: IS NULL and IS NOT NULL
- Wildcard characters: LIKE and NOT LIKE
- Subqueries: EXISTS and NOT EXISTS

## ► Conditions can be negated and combined with NOT AND OR

- Order of execution is: NOT, AND, OR
- Use parentheses to change priority if necessary

# WHERE Examples

- Which employees in Division 2 earn more than \$3,000?

c4-06.sql

```
SELECT EmployeeID, FirstName, LastName, CurrentSalary
FROM Employees
WHERE CurrentSalary > 3000
      AND DivisionID = 2;
```

Rows that satisfy  
both conditions are  
included in the result

EmployeeID	FirstName	LastName	CurrentSalary
6	Michael	Suyama	3600
8	Laura	Callahan	4000
9	Anne	Dodsworth	5200

# WHERE Examples

► **List the Employees with a salary between \$3,000 and \$5,000**

c4-07.sql

- Border values 3000 and 5000 included

```
SELECT EmployeeID, FirstName, LastName, CurrentSalary
FROM   Employees
WHERE  CurrentSalary BETWEEN 3000 AND 5000;
```

EmployeeID	FirstName	LastName	CurrentSalary
-----	-----	-----	-----
4	Margaret	Peacock	3000
5	Steven	Buchanan	5000
6	Michael	Suyama	3600
7	Robert	King	3000
8	Laura	Callahan	4000

# WHERE Examples

## ► List the customers in Norway and Sweden

c4-08.sql

```
SELECT CompanyName, Country
FROM   Customers
WHERE  Country IN ('Norway', 'Sweden');
```

Value list enclosed in parentheses, values separated by comma

CompanyName	Country
Berglunds snabbköp	Sweden
Folk och fä HB	Sweden
Norske Meierier	Norway
Santé Gourmet	Norway

# LIKE

- ▶ **LIKE is used for wildcard testing and uses the following special characters**

Character	Meaning
%	Any number (including zero) of any character
_	Any character, but exactly one occurrence

- ▶ **In a condition WHERE <column> LIKE <pattern>**

Pattern	The condition is true if the content of the column...
'A%'	starts with an A
'%B'	ends with a B
'%C%'	contains at least one C
'%DEF%'	contains the string DEF
'_G%'	has a G as its second character
'____'	is exactly three characters long
'____%'	is at least three characters long

# LIKE Examples

c4-09.sql

- List the employees whose last names start with *D*

SELECT EmployeeID	EmployeeID	LastName
, LastName	-----	-----
FROM Employees	1	Davolio
WHERE LastName LIKE 'D%';	9	Dodsworth

- List the employees whose last names end with *an*

SELECT EmployeeID	EmployeeID	LastName
, LastName	-----	-----
FROM Employees	5	Buchanan
WHERE LastName LIKE '%an';	8	Callahan

- List the employees whose last names have *a* as the second letter

SELECT EmployeeID	EmployeeID	LastName
, LastName	-----	-----
FROM Employees	8	Callahan
WHERE LastName LIKE '_a%';	1	Davolio

# Contents

---

- ▶ **SELECT Statement Structure**
- ▶ **Columns and Expressions**
- ▶ **WHERE Filter Conditions**

## **ORDER BY to Specify Sorting**

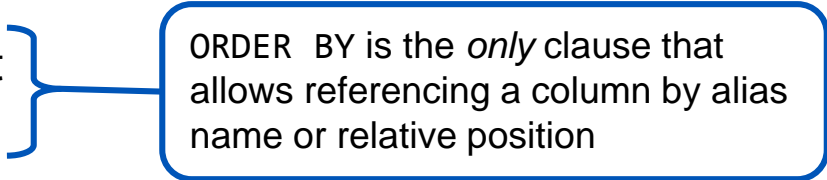
- ▶ **NULL Values**
- ▶ **DISTINCT**





# ORDER BY Clause

---

- ▶ **The sort order of the rows in the result set is undefined unless an ORDER BY clause is specified**
  - The rows may appear to be sorted, but this could be a *coincidence*
  - The only way to guarantee a sort order is to include an ORDER BY clause
- ▶ **The ORDER BY clause accepts**
  - Column name
  - Expression
  - Relative position in column list
  - Column alias

ORDER BY is the *only* clause that allows referencing a column by alias name or relative position
- ▶ **Sorting can be ascending (default or specified as ASC) or descending (specified as DESC)**
  - Applies to each individual column if multiple sort columns

# ORDER BY Examples

## ► Different ways of specifying ORDER BY

c4-10.sql

- All the clauses below give identical results

```
SELECT LastName
       ,FirstName
       ,CurrentSalary AS Salary
FROM   Employees
ORDER BY CurrentSalary DESC
       ,LastName      ASC
       ,FirstName      ASC;
```

```
.....
ORDER BY Salary DESC
       ,LastName
       ,FirstName;
```

```
.....
ORDER BY 3 DESC
       ,1
       ,2;
```

LastName	FirstName	Salary
-----	-----	-----
Fuller	Andrew	9000.0000
Leverling	Janet	6000.0000
Dodsworth	Anne	5200.0000
Buchanan	Steven	5000.0000
Callahan	Laura	4000.0000
Suyama	Michael	3600.0000
King	Robert	3000.0000
Peacock	Margaret	3000.0000
Davolio	Nancy	2000.0000

Columns separated by comma

# Contents

---

- ▶ **SELECT Statement Structure**
- ▶ **Columns and Expressions**
- ▶ **WHERE Filter Conditions**
- ▶ **ORDER BY to Specify Sorting**

## NULL Values

- ▶ **DISTINCT**



# NULL Values

---

- ▶ **NULL means *unknown* or *missing* value**
  - NULL is not the same as blank
  - NULL is not the same as zero
  - NULL is not the same as the text 'NULL '
- ▶ **Programmers need to be aware of NULL values and their behavior to avoid erroneous or unexpected results**
  - There are functions that can help in dealing with NULLs

# NULL Values

► **The following slides will demonstrate NULL value behavior using the Commission column in the Employees table**

- Three employees have a value in the column
- Six employees have NULL

c4-11.sql

```
SELECT FirstName, LastName, CurrentSalary, Commission
FROM   Employees;
```

FirstName	LastName	CurrentSalary	Commission
-----	-----	-----	-----
Nancy	Davolio	2000	
Andrew	Fuller	9000	
Janet	Leverling	6000	
Margaret	Peacock	3000	
Steven	Buchanan	5000	
Michael	Suyama	3600	200
Robert	King	3000	400
Laura	Callahan	4000	
Anne	Dodsworth	5200	300

# NULL Values in Expressions

- ▶ A NULL value in an expression causes the result to be NULL

c4-12.sql

```
SELECT FirstName, LastName, CurrentSalary, Commission
       ,CurrentSalary+Commission AS TotalPay
FROM   Employees;
```

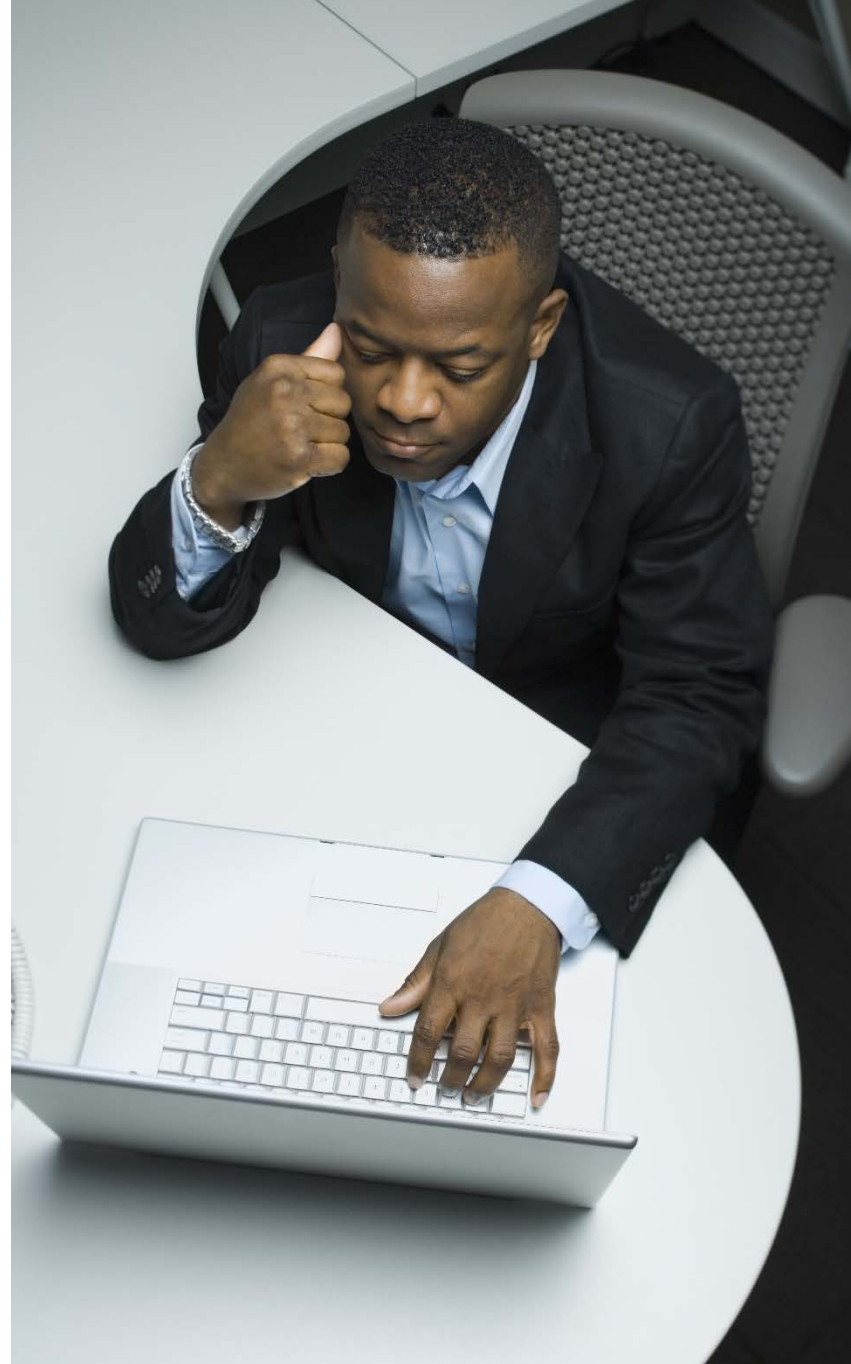
FirstName	LastName	CurrentSalary	Commission	TotalPay
-----	-----	-----	-----	-----
Nancy	Davolio	2000		
Andrew	Fuller	9000		
Janet	Leverling	6000		
Margaret	Peacock	3000		
Steven	Buchanan	5000		
Michael	Suyama	3600	200	3800
Robert	King	3000	400	3400
Laura	Callahan	4000		
Anne	Dodsworth	5200	300	5500



# NULLs in Conditions

---

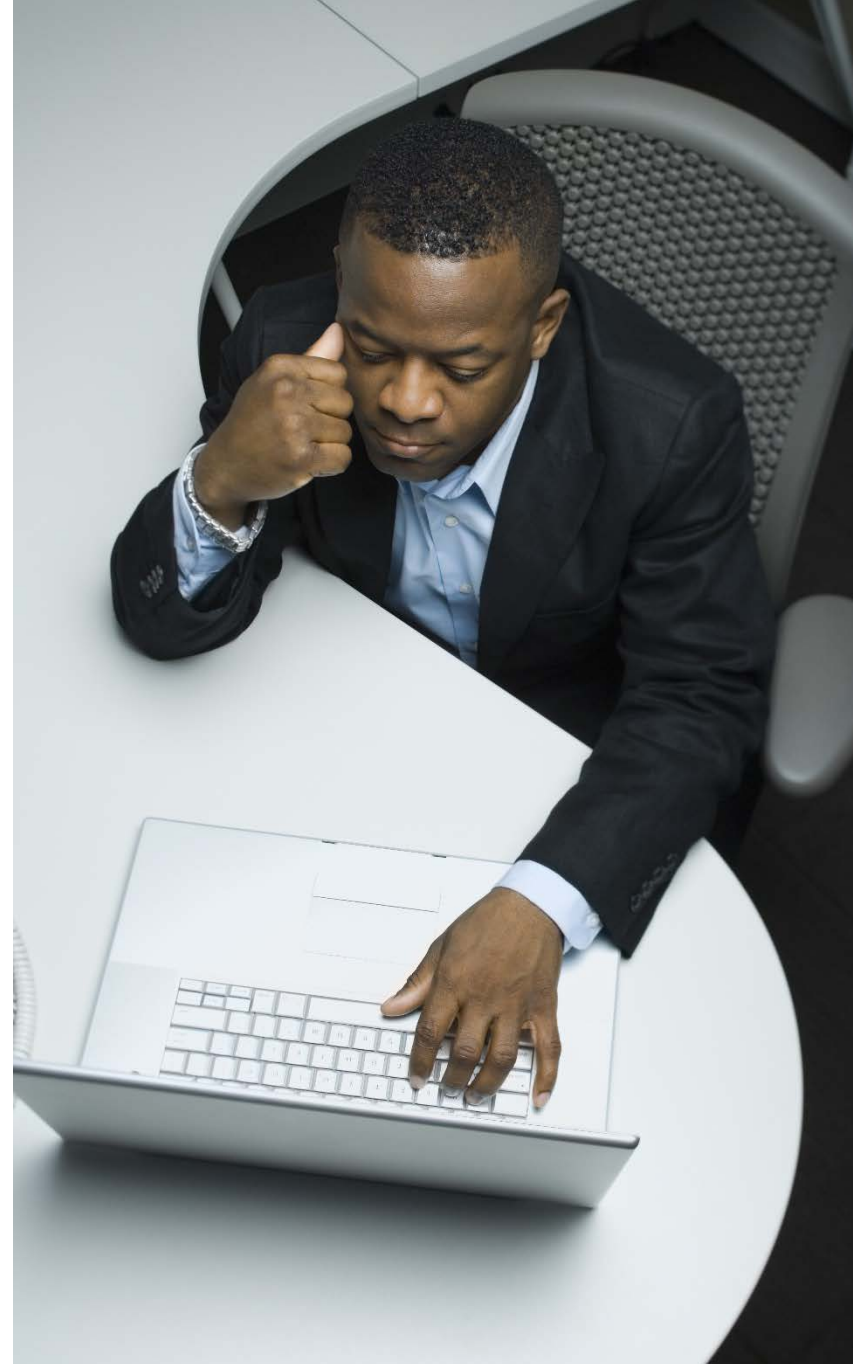
- ▶ **When an unknown value is tested in a condition, the result of the test is also unknown**
- ▶ **A WHERE condition must decide whether or not to return the row**
  - If it is not certain that the condition is true, it is assumed to be false
- ▶ **Consequently, a condition involving NULL is evaluated as false**
- ▶ **The test operator IS NULL returns true when the operand is NULL**
- ▶ **The test operator IS NOT NULL returns true when the operand is not NULL**
- ▶ **NULL is not equal to any value**
- ▶ **NULL is not different from any value**



# NULLs in Conditions

---

- ▶ NULL is not greater or less than any value
- ▶ NULL is not equal to NULL
- ▶ NULL is not different from NULL
- ▶ NULL is not greater or less than NULL
- ▶ BETWEEN and NOT BETWEEN are false if one of the limits is NULL
- ▶ IN will not give a match on an element that is NULL
- ▶ NOT IN is false if at least one of the elements is NULL
- ▶ If a *<condition>* is evaluated as false because of NULL, then NOT *<condition>* will also be evaluated as false



# Examples of NULLs in Conditions

## ► Examples

c4-14.sql

```
SELECT LastName
       ,Commission
FROM   Employees
WHERE  Commission < 400;
```

LastName	Commission
-----	-----
Suyama	200
Dodsworth	300

```
.....
WHERE  NOT Commission < 400;
```

LastName	Commission
-----	-----
King	400

```
.....
WHERE  Commission <> 300;
```

LastName	Commission
-----	-----
Suyama	200
King	400

# Examples of NULLs in Conditions

## ► More examples

c4-15.sql

```
SELECT LastName
       ,Commission
FROM   Employees
WHERE  Commission = NULL;

.....

WHERE  Commission <> NULL;

.....

WHERE  Commission = Commission;
```

LastName	Commission
-----	-----
LastName	Commission
-----	-----
LastName	Commission
-----	-----
Suyama	200
King	400
Dodsworth	300

# Examples of NULLs in Conditions

## ► More examples

c4-16.sql

```
SELECT LastName
       ,Commission
FROM   Employees
WHERE  Commission IS NULL;
```

LastName	Commission
-----	-----
Davolio	
Fuller	
Leverling	
Peacock	
Buchanan	
Callahan	

```
.....
WHERE  Commission IS NOT NULL;
```

LastName	Commission
-----	-----
Suyama	200
King	400
Dodsworth	300

# Example of NULL and IN

- **IN will not give a match on a NULL value**

c4-17.sql

```
SELECT LastName
       ,ReportsTo
FROM   Employees
WHERE  ReportsTo IN (2,NULL);
```

is equivalent to

```
SELECT LastName
       ,ReportsTo
FROM   Employees
WHERE  ReportsTo = 2
      OR ReportsTo = NULL;
```

LastName	ReportsTo
-----	-----
Davolio	2
Leverling	2
Peacock	2
Buchanan	2
Callahan	2

- **Andrew Fuller reports to NULL and is not included in the result, since NULL is not equal to NULL**

# Example of NULL and NOT IN

- ▶ NOT IN will not be true if there is a NULL among the values

c4-18.sql

```
SELECT LastName          LastName   ReportsTo
       ,ReportsTo        -----
FROM   Employees
WHERE  ReportsTo NOT IN (2,NULL);
```

is equivalent to

```
SELECT LastName
       ,ReportsTo
FROM   Employees
WHERE  ReportsTo <> 2
      AND ReportsTo <> NULL;
```

- ▶ NULL is not different from NULL, so the condition is never satisfied



# NULLs and Sorting

- ▶ The standard specifies that NULL values should be sorted together, but does not specify *where* in the sort order
- ▶ Different products have made different decisions
  - In PostgreSQL and Oracle, NULL sorts as the highest value
    - But NULLS FIRST or NULLS LAST can override
  - In SQL Server, NULL sorts as the lowest value

c4-19.sql

## ▶ Example

```
SELECT LastName
       ,Commission
FROM   Employees
ORDER BY Commission
       ,LastName;
```

### SQL Server

LastName	Commission
-----	-----
Buchanan	
Callahan	
Davolio	
Fuller	
Leverling	
Peacock	
Suyama	200
Dodsworth	300
King	400

NULL sorts  
as lowest  
value

### PostgreSQL and Oracle

LastName	Commission
-----	-----
Suyama	200
Dodsworth	300
King	400
Buchanan	
Callahan	
Davolio	
Fuller	
Leverling	
Peacock	

NULL sorts  
as highest  
value

# Contents

---

- ▶ **SELECT Statement Structure**
- ▶ **Columns and Expressions**
- ▶ **WHERE Filter Conditions**
- ▶ **ORDER BY to Specify Sorting**
- ▶ **NULL Values**

## **DISTINCT**



# Duplicate Rows

- ▶ According to relational theory, a table cannot have duplicate rows
- ▶ SQL permits duplicate rows
  - In a stored table if primary key is not defined
  - In the result of a query
- ▶ Use **SELECT DISTINCT** to suppress duplicates
- ▶ Example

c4-20.sql

```
SELECT DISTINCT
    TitleOfCourtesy
FROM    Employees
ORDER BY TitleOfCourtesy;
```

```
TitleOfCourtesy
-----
Dr.
Mr.
Mrs.
Ms.
```

# Duplicate Rows

## ► Example

- DISTINCT applies to the *complete* row with *all* its columns

c4-21.sql

```
SELECT DISTINCT
    DivisionID
    ,TitleOfCourtesy
FROM    Employees
ORDER BY DivisionID
    ,TitleOfCourtesy;
```

DivisionID	TitleOfCourtesy
-----	-----
1	Dr.
1	Mr.
1	Ms.
2	Mr.
2	Mrs.
2	Ms.

## Hands-On Exercise 4.2

In your Exercise Manual, please refer to  
**Hands-On Exercise 4.2: WHERE and ORDER BY**

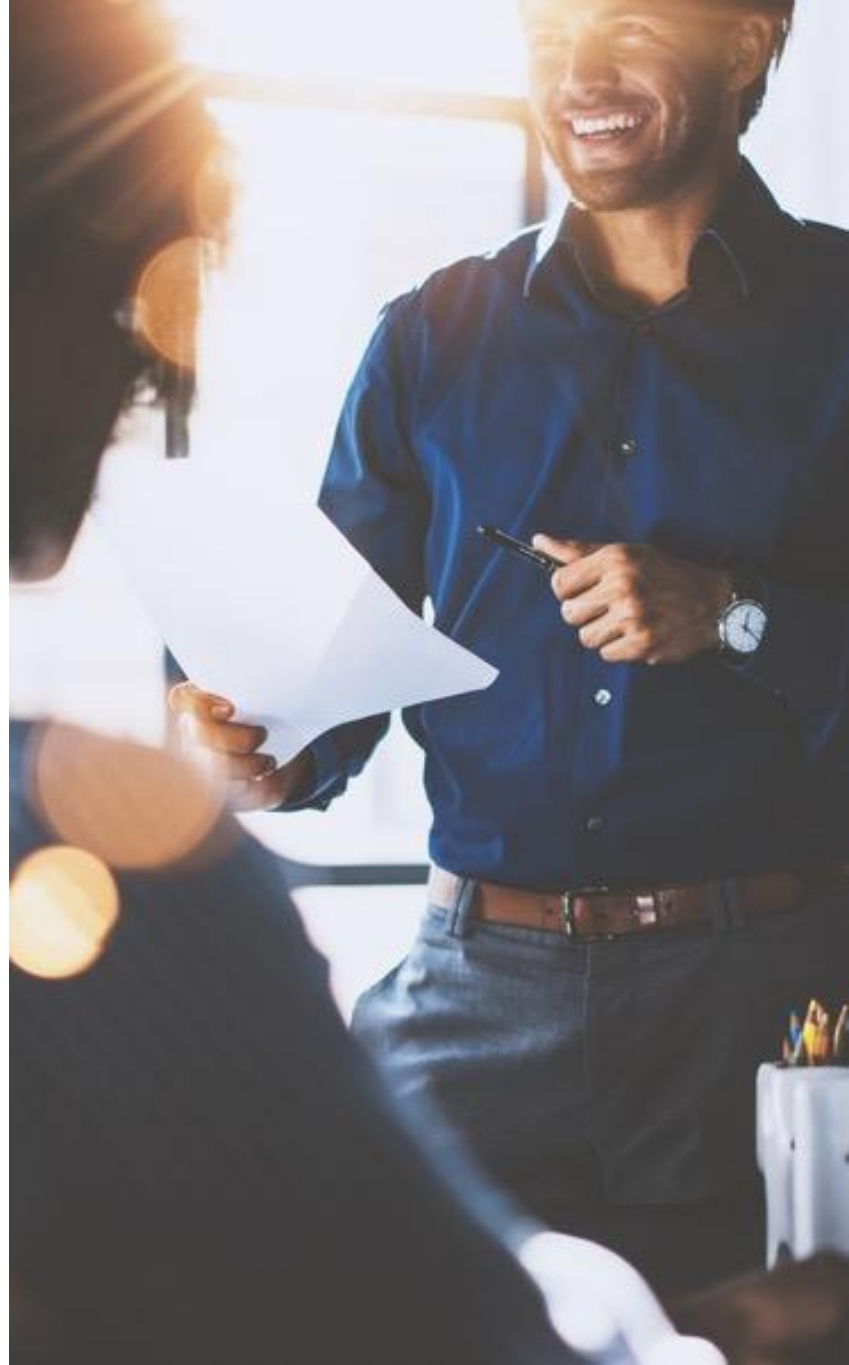
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Describe the basic structure of a **SELECT** statement
- ▶ Retrieve columns and expressions
- ▶ Apply filter conditions
- ▶ Sort the result set
- ▶ Describe issues with **NULL** values
- ▶ Use **DISTINCT** to suppress duplicate rows





# Chapter 5

## Joins



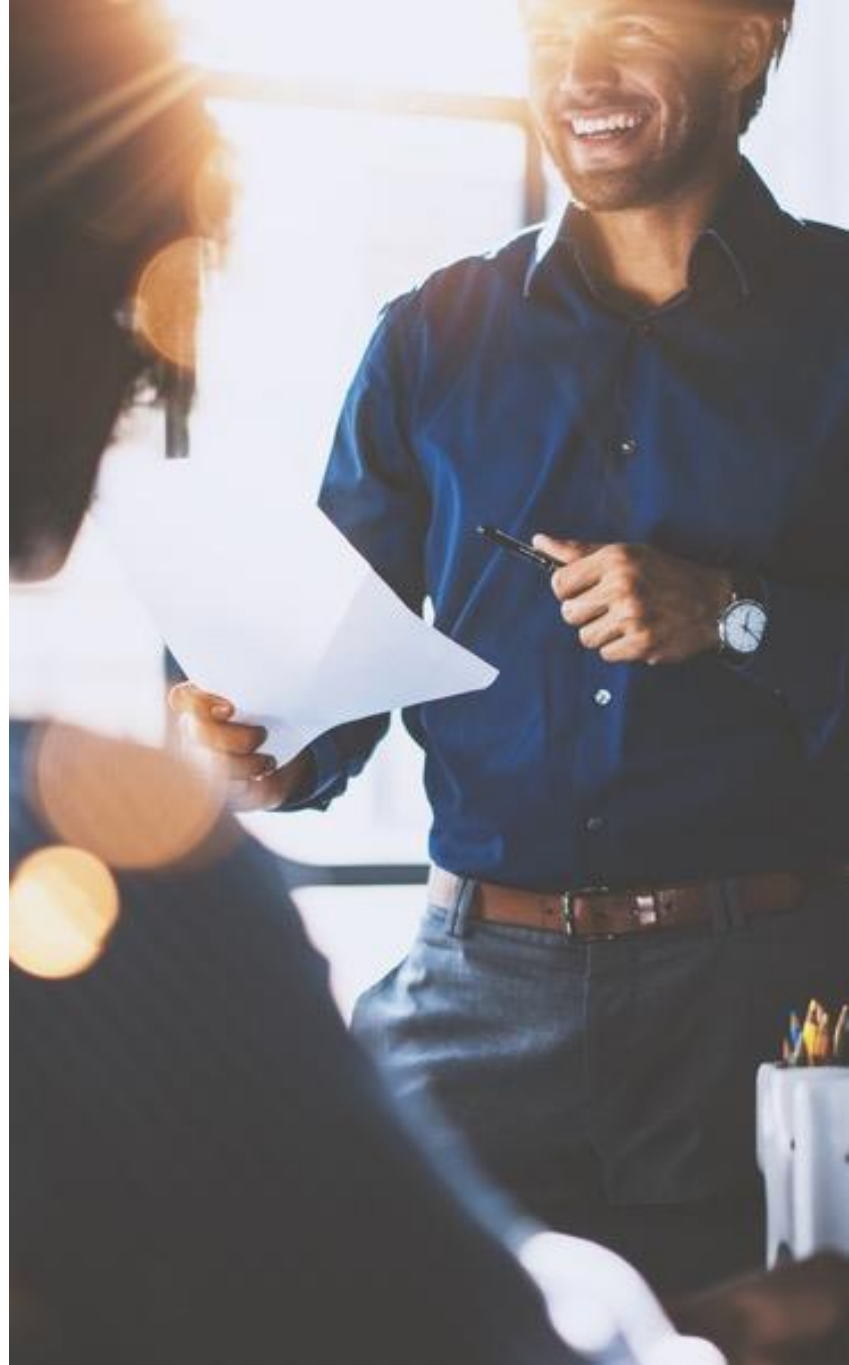
LEARNING TREE™  
INTERNATIONAL



# Objectives

---

- ▶ **Use joins to retrieve data from multiple tables**
- ▶ **Learn the difference between inner, outer, and cross joins**
- ▶ **Join a table to itself**



# Contents

---

## Inner Join

- ▶ Outer Join
- ▶ Cross Join
- ▶ Self-Join



# Joining Multiple Tables

---

- ▶ **A common requirement is to combine information from multiple tables**
  - In a normalized database, you will rarely find all the desired information in a single table
- ▶ **Related rows from different tables can be joined together**
  - Most commonly related through foreign key/primary key pairs
- ▶ **A *join condition* is required to specify how the rows of the different tables relate to each other**
  - The join condition is specified with the keyword ON
- ▶ **Basic join syntax**  
`FROM <table name> JOIN <table name> ON <join condition>`
- ▶ **There are three different varieties of joins: INNER, OUTER, and CROSS**
  - We will cover one at a time
- ▶ **Inner joins can be specified with the keyword INNER JOIN**
  - This is the default, so the word INNER may be omitted

# Join Example

## ► Consider the Divisions and Departments tables

Divisions

DivisionID	DivisionName
1	America
2	Europe
3	Pacific

Departments

DivisionID	DepartmentID	DepartmentName
1	100	Accounting
1	200	Sales
1	300	Research
2	100	Sales
2	200	Accounting
2	300	Research
3	100	Administration

## ► Each Department belongs to a particular Division, and we would like to list the Departments and include the names of their Divisions

DivisionID	DivisionName	DepartmentID	DepartmentName
1	America	100	Accounting
1	America	200	Sales
1	America	300	Research
2	Europe	100	Sales
2	Europe	200	Accounting
2	Europe	300	Research
3	Pacific	100	Administration

# Join Example

- ▶ The following query will produce the result on the previous slide

```
SELECT Departments.DivisionID
      ,Divisions.DivisionName
      ,Departments.DepartmentID
      ,Departments.DepartmentName
FROM   Departments
JOIN   Divisions
ON     Departments.DivisionID = Divisions.DivisionID;
```

c5-01.sql

- ▶ **Explanation**

- A join returns columns from more than one table
  - In this case, a join is required because we need columns from two tables
- The ON condition specifies how the rows in the tables relate to each other
- The column names have prefixes to specify the table in which each column is located
  - Prefixes will be discussed later

# Another Join Example

► **For each order detail, list the order ID, product ID, product name, unit price, and quantity**

- ProductName is a column in the Products table
- The other columns are located in the OrderDetails table

```
SELECT OrderDetails.OrderID
       ,OrderDetails.ProductID
       ,Products.ProductName
       ,OrderDetails.UnitPrice
       ,OrderDetails.Quantity
FROM   OrderDetails
JOIN   Products
ON     OrderDetails.ProductID = Products.ProductID;
```

c5-02.sql

OrderID	ProductID	ProductName	UnitPrice	Quantity
10248	11	Queso Cabrales	14	12
10248	42	Singaporean Hokkien Fried Mee	9.8	10
10248	72	Mozzarella di Giovanni	34.8	5
10249	14	Tofu	18.6	9
.....				

# Column Names

---

- ▶ **A column name that occurs in more than one of the tables must have the table name as a prefix so as not to be ambiguous**
- ▶ **Prefixing all column names is strongly recommended, even when it is not strictly necessary**
  - Improves readability
  - No risk of becoming ambiguous if
    - New tables are added to the query
    - New columns are added to tables



# Table Alias

---

## ► A table can have an alias name

- Also called *range variable* or *correlation name*
- Like column alias names, the keyword AS is specified in the standard, but most products do not require it
  - Oracle does *not allow* the keyword AS with table alias names
  - For this reason, we will *omit* the AS with table alias names in the course examples

## ► The alias *replaces* the real table name *within* the query

- The alias must be used as a prefix instead of the real table name
- A table alias is useful to reduce typing and improve readability

## ► Use alias in the previous example

c5-03.sql

```
SELECT od.OrderID, od.ProductID, p.ProductName
       ,od.UnitPrice, od.Quantity
FROM   OrderDetails od
JOIN   Products      p
ON     od.ProductID = p.ProductID;
```

# Join and WHERE

## ► It is possible to combine a WHERE condition with a join

- Example: Restrict the previous example to include only products in Category 1

```
SELECT od.OrderID
       ,od.ProductID
       ,p.ProductName
       ,od.UnitPrice
       ,od.Quantity
```

```
FROM   OrderDetails od
JOIN    Products      p
ON      od.ProductID = p.ProductID
WHERE   p.CategoryID = 1;
```

c5-04.sql

Join condition

Filter condition

OrderID	ProductID	ProductName	UnitPrice	Quantity
10253	39	Chartreuse verte	14.4	42
10254	24	Guaraná Fantástica	3.6	15
10255	2	Chang	15.2	20
10257	39	Chartreuse verte	14.4	6
.....				

# Join and Sorting

## ► It is impossible to predict the sort order of a join result

- Unless ORDER BY is specified
- Example: Sort the result of the previous query by Product ID and Order ID

```
SELECT od.OrderID, od.ProductID, p.ProductName
       ,od.UnitPrice, od.Quantity
FROM   OrderDetails od
JOIN   Products p
ON     od.ProductID = p.ProductID
WHERE  p.CategoryID = 1
ORDER BY p.ProductID, od.OrderId;
```

c5-05.sql

OrderID	ProductID	ProductName	UnitPrice	Quantity
10285	1	Chai	14.4	45
10294	1	Chai	14.4	18
11070	1	Chai	18	40
10255	2	Chang	15.2	20
11077	2	Chang	19	24
10254	24	Guaraná Fantástica	3.6	15

# More Than Two Tables

---

- ▶ **There may be more than two tables in a join**
  - All tables must have a join condition
  - The keywords JOIN and ON are repeated for each additional table
- ▶ **Example: Include customer name**
  - Customer name is a column in the Customer table
  - The relationship between OrderDetails and Customers is via Orders

```
SELECT od.OrderID
       ,c.CompanyName
       ,od.ProductID
       ,p.ProductName
       ,od.UnitPrice
       ,od.Quantity
FROM   OrderDetails od
JOIN   Products p
ON     od.ProductID = p.ProductID
JOIN   Orders o
ON     o.OrderID = od.orderID
JOIN   Customers c
ON     c.CustomerID = o.CustomerID
ORDER BY od.OrderID, od.ProductID;
```

c5-06.sql

# More Than Two Tables

## ► Result of the query

c5-06.sql

OrderID	CompanyName	ProductID	ProductName	UnitPrice	Quantity
10248	Vins et alcools ...	11	Queso Cabrales	14.0000	12
10248	Vins et alcools ...	42	Singaporean ...	9.8000	10
10248	Vins et alcools ...	72	Mozzarella di ...	34.8000	5
10249	Toms Spezialitäten	14	Tofu	18.6000	9
.....					

# Old Nonstandard Syntax

---

- ▶ **There is an alternative syntax that does not follow the standard**
  - Introduced by the first IBM SQL dialect long before the standard existed
  - Still supported by most products, for backward compatibility
- ▶ **As a programmer, you are likely to encounter old programs that use the old syntax, so you need to be aware of it**
  - Recommended to use standard syntax when developing new programs
- ▶ **Join syntax**  
FROM <table name>,...,<table name>  
WHERE <join condition>
- ▶ **This syntax uses the WHERE clause for two semantically different purposes**
  - Join conditions
  - Filter conditions (restrictions)

# Old Nonstandard Syntax

## ► The four-way join using the nonstandard syntax

c5-07.sql

```
SELECT od.OrderID
       ,c.CompanyName
       ,od.ProductID
       ,p.ProductName
       ,od.UnitPrice
       ,od.Quantity
FROM   OrderDetails od
       ,Products p
       ,Orders o
       ,Customers c
WHERE  od.ProductID = p.ProductID
       AND o.OrderID = od.orderID
       AND c.CustomerID =
o.CustomerID;
```



## Hands-On Exercise 5.1

In your Exercise Manual, please refer to Hands-On Exercise 5.1: Inner Join

- ▶ Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`
- ▶ You should save your own solutions to the exercises, since some of the exercises build on previous exercises
- ▶ However, if you have not saved your own solutions, you can always copy from the provided solution files

# Contents

---

- ▶ Inner Join

## Outer Join

- ▶ Cross Join
- ▶ Self-Join



# Inner and Outer Join

---

- ▶ **The joins we have seen so far have all been *inner* joins**
  - The result excludes rows that do *not* satisfy the join condition; i.e., have *no* matching rows in the other table
- ▶ **Inner joins can be specified with the keyword `INNER JOIN`**
  - This is the default, so the word `INNER` may be omitted, as we have seen
- ▶ **It may be desirable to include rows from one table that have no matching row in the other table**
  - This is called an *outer join*
  - Columns will contain `NULL` when there are no matching rows
- ▶ **Common situations where there are no matching rows**
  - A primary key value with no corresponding foreign key values
    - There should never be foreign key values with no corresponding primary key value, because this would violate the referential integrity rule
  - A foreign key with `NULL` value

# Outer Join Syntax

- ▶ **Outer joins are specified with the keyword OUTER preceded by the qualifier LEFT, RIGHT, or FULL**
  - Since LEFT, RIGHT, and FULL are used only with outer joins, the keyword OUTER is redundant and may be omitted
  - For clarity, it is most common to include OUTER

Syntax	Meaning
<b>table1</b> LEFT OUTER JOIN <b>table2</b>	<b>Table1</b> (to the <i>left</i> of the JOIN keyword) is the most important table (the <i>driving</i> table) <b>Table2</b> is the <i>driven</i> table All rows from <b>table1</b> will be included, including those that have no matching rows in <b>table2</b>
<b>table1</b> RIGHT OUTER JOIN <b>table2</b>	<b>Table2</b> (to the <i>right</i> of the JOIN keyword) is the most important table (the <i>driving</i> table) <b>Table1</b> is the <i>driven</i> table All rows from <b>table2</b> will be included, including those that have no matching rows in <b>table1</b>
<b>table1</b> FULL OUTER JOIN <b>table2</b>	<b>Both</b> tables are equally important All rows from each table will be included

# Outer Join Example

## ► List the conference rooms and their reservations

- Notice that rooms without reservations are missing (inner join)

```
SELECT c.RoomID, c.Name, r.StartTime
FROM   ConferenceRooms c
JOIN   Reservations r
ON     r.RoomID = c.RoomID;
```

c5-08.sql

RoomID	Name	StartTime
101	Auditorium	2000-11-20 08:00
101	Auditorium	2000-12-22 09:00
300	3rd Floor Small room	2000-12-22 09:00
101	Auditorium	2000-12-22 11:00
220	2nd Floor Big room	2000-12-22 10:30
220	2nd Floor Big room	2000-12-22 08:00
220	2nd Floor Big room	2000-12-22 10:00
220	2nd Floor Big room	2000-12-15 09:30
220	2nd Floor Big room	2000-12-15 11:00
300	3rd Floor Small room	2000-12-15 11:00
101	Auditorium	2000-12-22 08:00
220	2nd Floor Big room	2000-12-22 16:30

# Outer Join Example

## ► Include the rooms without reservations (outer join)

- If there are no reservations for a given room, only the room information is shown
- Missing information is shown as NULL

## ► Notice that the ordering has changed

- This is because the outer join forces the driving table
- As with all queries, to determine a specific sort order, an ORDER BY clause is required

```
SELECT c.RoomID, c.Name, r.StartTime
FROM   ConferenceRooms c
LEFT OUTER JOIN
      Reservations r
ON     r.RoomID = c.RoomID;
```

c5-09.sql

## ► The ConferenceRooms table occurs to the left of (i.e. before) the JOIN keyword, and is the most important table

# Outer Join Example

## ► Result of the query

c5-09.sql

RoomID	Name	StartTime
101	Auditorium	2000-11-20 08:00
101	Auditorium	2000-12-22 09:00
101	Auditorium	2000-12-22 11:00
101	Auditorium	2000-12-22 08:00
220	2nd Floor Big room	2000-12-22 10:30
220	2nd Floor Big room	2000-12-22 08:00
220	2nd Floor Big room	2000-12-22 10:00
220	2nd Floor Big room	2000-12-15 09:30
220	2nd Floor Big room	2000-12-15 11:00
220	2nd Floor Big room	2000-12-22 16:30
225	2nd Floor Small room	
300	3rd Floor Small room	2000-12-22 09:00
300	3rd Floor Small room	2000-12-15 11:00
310	3rd Floor Tiny room	

Room with no  
reservations

Room with no  
reservations



## Hands-On Exercise 5.2

**In your Exercise Manual, please refer to  
Hands-On Exercise 5.2: Outer Join**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**

# Contents

---

- ▶ Inner Join
- ▶ Outer Join

## Cross Join

- ▶ Self-Join



# Cross Join

► A *cross join* has no join condition (ON clause)

- Each row in one table is paired to every row in the other table
- The keyword for a cross join is CROSS JOIN

	*		=		
A		1		A	1
B		2		A	2
		3		A	3
				B	1
				B	2
				B	3

► A cross join is also called a *Cartesian product*

► Cross joins are not frequently used

# Cross Join Example

## ► List all employees with all divisions

```
SELECT e.FirstName
       ,e.LastName
       ,d.DivisionName
FROM   Employees e
CROSS JOIN
       Divisions d;
```

c5-10.sql

No join condition

- Each employee repeated three times
  - For each of the three divisions
- Each division repeated nine times
  - For each of the nine employees

FirstName	LastName	DivisionName
-----	-----	-----
Nancy	Davolio	America
Andrew	Fuller	America
Janet	Leverling	America
Margaret	Peacock	America
Steven	Buchanan	America
Michael	Suyama	America
Robert	King	America
Laura	Callahan	America
Anne	Dodsworth	America
Nancy	Davolio	Europe
Andrew	Fuller	Europe
Janet	Leverling	Europe
Margaret	Peacock	Europe
Steven	Buchanan	Europe
Michael	Suyama	Europe
Robert	King	Europe
Laura	Callahan	Europe
Anne	Dodsworth	Europe
Nancy	Davolio	Pacific
Andrew	Fuller	Pacific
Janet	Leverling	Pacific
Margaret	Peacock	Pacific
Steven	Buchanan	Pacific
Michael	Suyama	Pacific
Robert	King	Pacific
Laura	Callahan	Pacific
Anne	Dodsworth	Pacific

# Contents

---

- ▶ Inner Join
- ▶ Outer Join
- ▶ Cross Join

## Self-Join





# Self-Join

## ► A table can be joined to itself

- The query “sees” two identical copies of the table
- Table aliases must be used to distinguish between the two

## ► Example of self-join

- For each employee, list the name of the employee and the name of the manager that he or she reports to
- ReportsTo is a foreign key that references the manager’s EmployeeID
- Outer join is required in order to include employees who have no manager (i.e., NULL in the ReportsTo column)

```
SELECT e.EmployeeID
       ,e.LastName
       ,e.ReportsTo
       ,m.LastName AS Manager
FROM   Employees e
LEFT OUTER JOIN
       Employees m
ON     e.ReportsTo = m.EmployeeID;
```

c5-11.sql

# Self-Join

- These are the two “copies” of the table

Employees

EmployeeID	LastName	ReportsTo
1	Davolio	2
2	Fuller	
3	Leverling	2
4	Peacock	2
5	Buchanan	2
6	Suyama	5
7	King	5
8	Callahan	2
9	Dodsworth	5

Foreign key

Managers

EmployeeID	LastName	ReportsTo
1	Davolio	2
2	Fuller	
3	Leverling	2
4	Peacock	2
5	Buchanan	2
6	Suyama	5
7	King	5
8	Callahan	2
9	Dodsworth	5

Primary key

- This is the result of the self-join query

EmployeeID	LastName	ReportsTo	Manager
1	Davolio	2	Fuller
2	Fuller		
3	Leverling	2	Fuller
4	Peacock	2	Fuller
5	Buchanan	2	Fuller
6	Suyama	5	Buchanan
7	King	5	Buchanan
8	Callahan	2	Fuller
9	Dodsworth	5	Buchanan

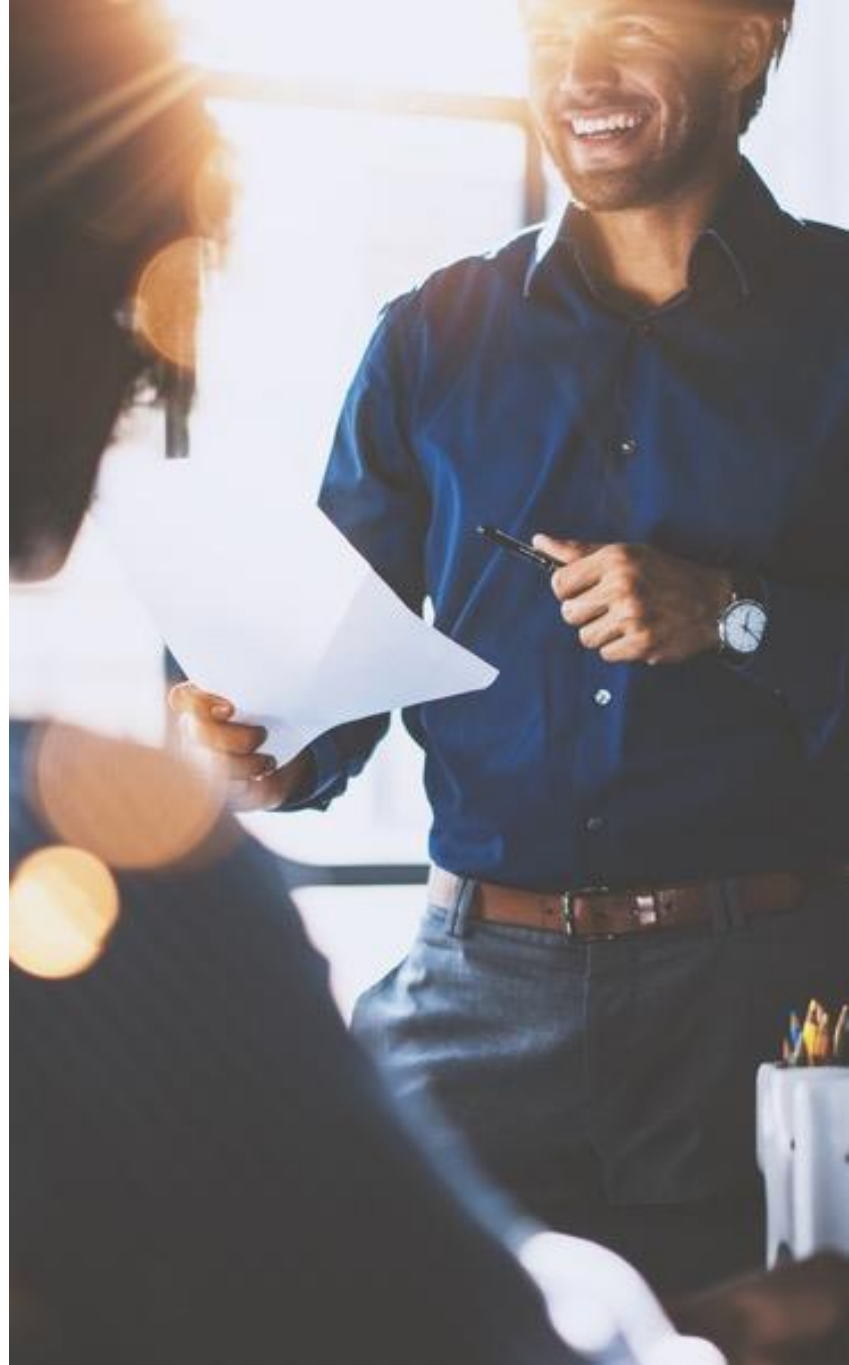
c5-11.sql



# Objectives

---

- ▶ **Use joins to retrieve data from multiple tables**
- ▶ **Learn the difference between inner, outer, and cross joins**
- ▶ **Join a table to itself**



# Chapter 6

## Set Operators

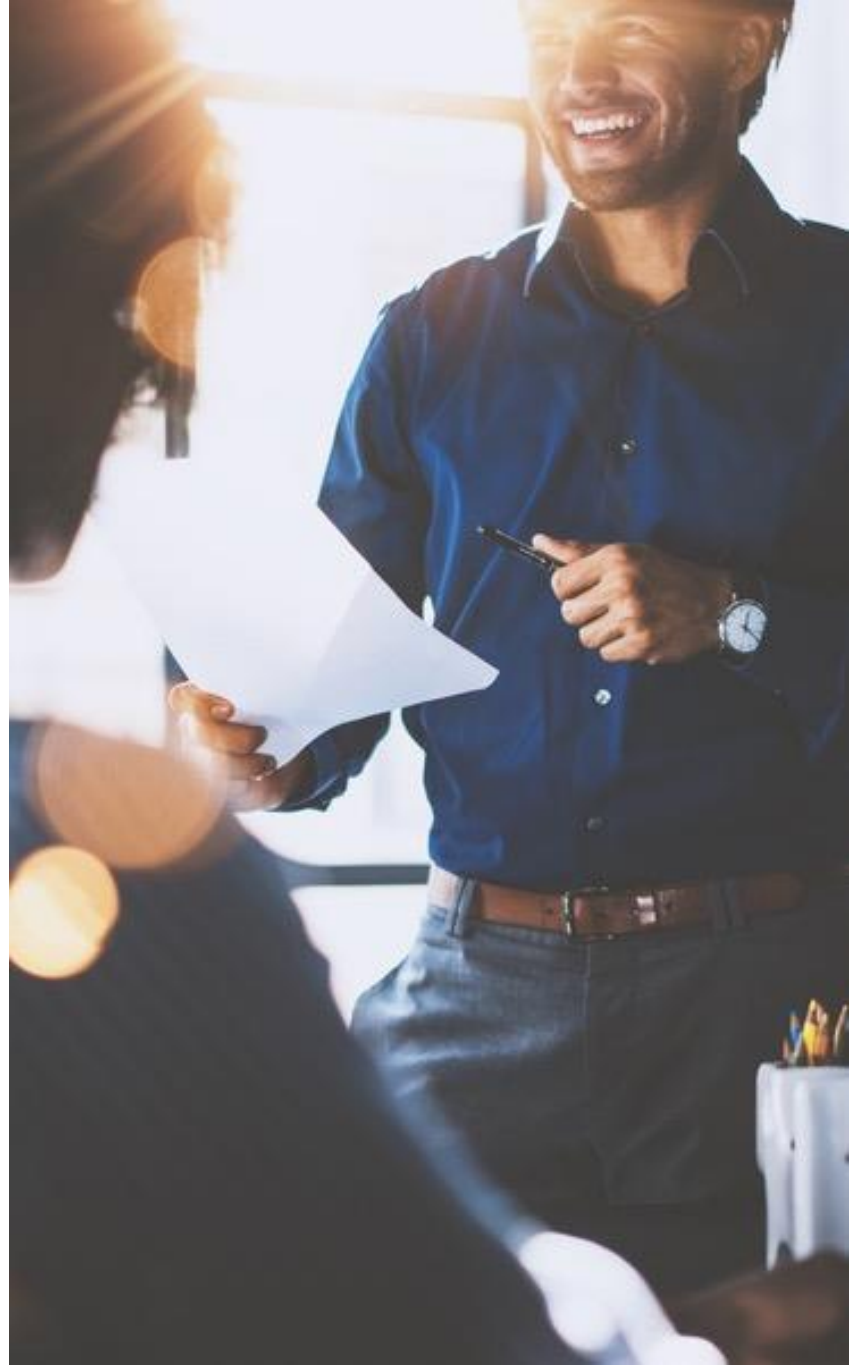


LEARNING TREE™  
INTERNATIONAL

# Objectives

---

- ▶ Use the set operators UNION, INTERSECT, and EXCEPT
- ▶ Describe the rules for set compatibility





# Contents

---

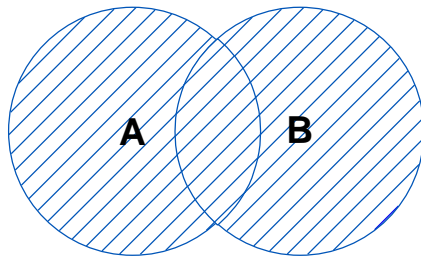
## Set Operators

### ► Set Compatibility

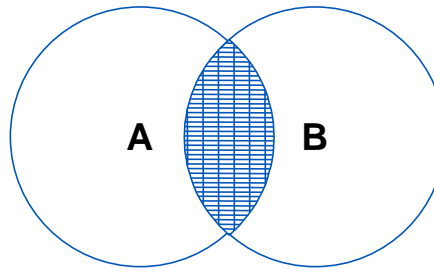


# Set Operators

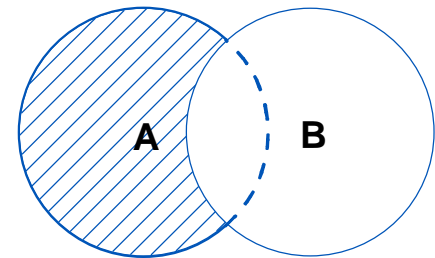
- ▶ A set operator combines the result of two queries
- ▶ The set operators are
  - UNION
  - INTERSECT
  - EXCEPT (also known as MINUS or DIFFERENCE)



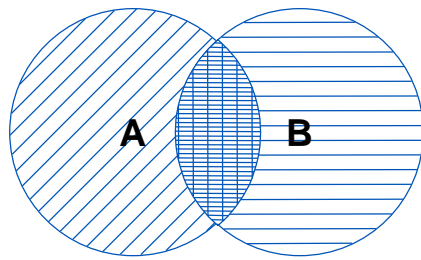
UNION



INTERSECT



EXCEPT



UNION ALL

# UNION

---

- ▶ **Syntax:** `<query1> UNION <query2> ORDER BY <expression>;`
- ▶ **Returns the rows from <query1> along with the rows from <query2>**
  - It is possible to have more than two queries  
`<query1> UNION <query2> UNION <query3> ...`
- ▶ **The first query determines the column names in the result**
  - Column or alias name
  - Alias names in the second (or third, etc.) query are ignored
- ▶ **ORDER BY is applied after the last query**
  - Each individual query may not do ORDER BY
  - ORDER BY references the column names in the final result as determined by the first query

# UNION vs. UNION ALL

---

- ▶ **UNION suppresses duplicate rows**
  - Identical rows are returned only once
- ▶ **Duplicate suppression is done by an implicit DISTINCT**
  - Duplicates within each query will be suppressed as well as duplicates across queries
- ▶ **UNION ALL does not suppress duplicates**



If you know that there will be no duplicates, use UNION ALL for better performance

---



# UNION Examples

► **List the companies with which we do business; i.e., suppliers and customers**

- If a company is both supplier and customer, it should be listed only once

```
SELECT CompanyName, City, Country
FROM Customers
```

c6-01.sql

**UNION**

```
SELECT CompanyName, City, Country
FROM Suppliers
ORDER BY CompanyName;
```

Sorts the result of the UNION

CompanyName	City	Country
Alfreds Futterkiste	Berlin	Germany
Ana Trujillo Emparedados y helados	México D.F.	Mexico
.....	.....	.....
Lyngbysild	Lyngby	Denmark
.....	.....	.....
Zaanse Snoepfabriek	Zaandam	Netherlands

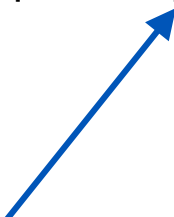
# UNION Examples

## ► List the names of persons we deal with

- Supplier and customer contacts as well as employees

```
SELECT ContactName AS Name
FROM   Customers
UNION
SELECT ContactName
FROM   Suppliers
UNION
SELECT FirstName || ' ' ||
LastName
FROM   Employees;
```

SQL Server uses +  
instead of ||



c6-02.sql

Name

-----  
Alejandra Camino  
Alexander Feuer  
Ana Trujillo  
Anabela Domingues  
André Fonseca  
Andrew Fuller  
Ann Devon  
Anne Dodsworth  
Anne Heikkonen  
Annette Roulet  
Antonio del Valle Saavedra  
Antonio Moreno

.....

# INTERSECT

---

- ▶ Returns the rows that occur in both queries
- ▶ Example: Find the Customers who are also Suppliers

```
SELECT CompanyName
FROM   Customers
INTERSECT
SELECT CompanyName
FROM   Suppliers;
```

```
CompanyName
-----
Lyngbysild
Norske Meierier
```

c6-03.sql

# EXCEPT

- ▶ Returns rows from one query except those that also occur in the other query
- ▶ Example: Find the Customers who have no Orders

c6-04.sql

```
SELECT CustomerID
FROM   Customers
EXCEPT
SELECT CustomerID
FROM   Orders;
```

```
CustomerID
-----
FISSA
LYBYS
NOMEI
PARIS
```

- ▶ **EXCEPT is the standard keyword**
  - Oracle uses the keyword MINUS instead

# Contents

---

## ► Set Operators

## Set Compatibility



# Set Compatibility

---

- ▶ **The set operators test the elements in the sets for equality**
  - In this context, “elements in sets” means “rows in query results”
- ▶ **The rows from the queries must be *set compatible***
  - Also called *union compatible*
  - Same number of columns
  - Columns must have same or compatible data type
- ▶ **If the data types are not the same, implicit conversion may take place**
  - Depending on product and version
  - If conversion is not possible, the query will fail





## Hands-On Exercise 6.1

**In your Exercise Manual, please refer to Hands-On Exercise 6.1: Set Operators**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**

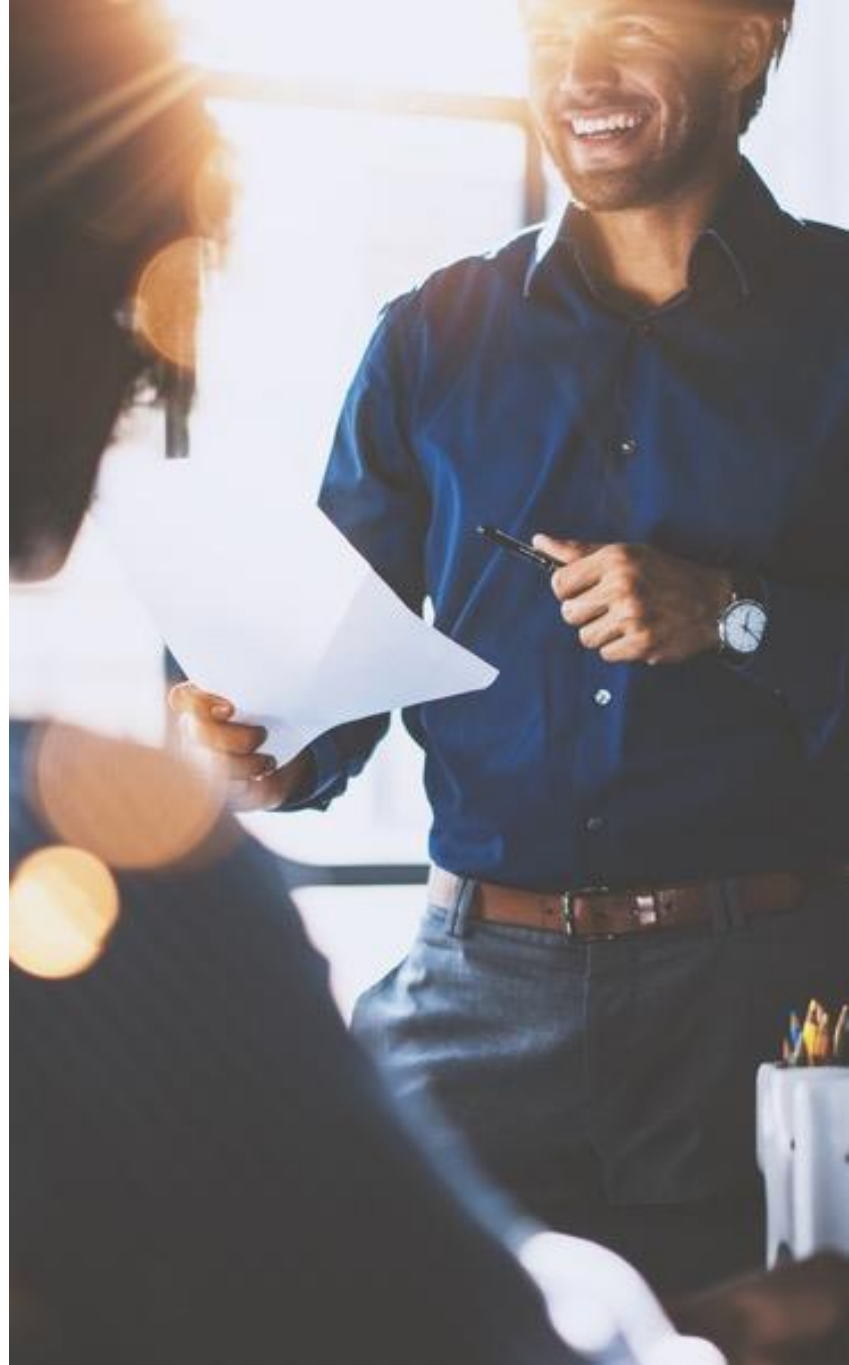




# Objectives

---

- ▶ Use the set operators UNION, INTERSECT, and EXCEPT
- ▶ Describe the rules for set compatibility



# Chapter 7

## Row Functions

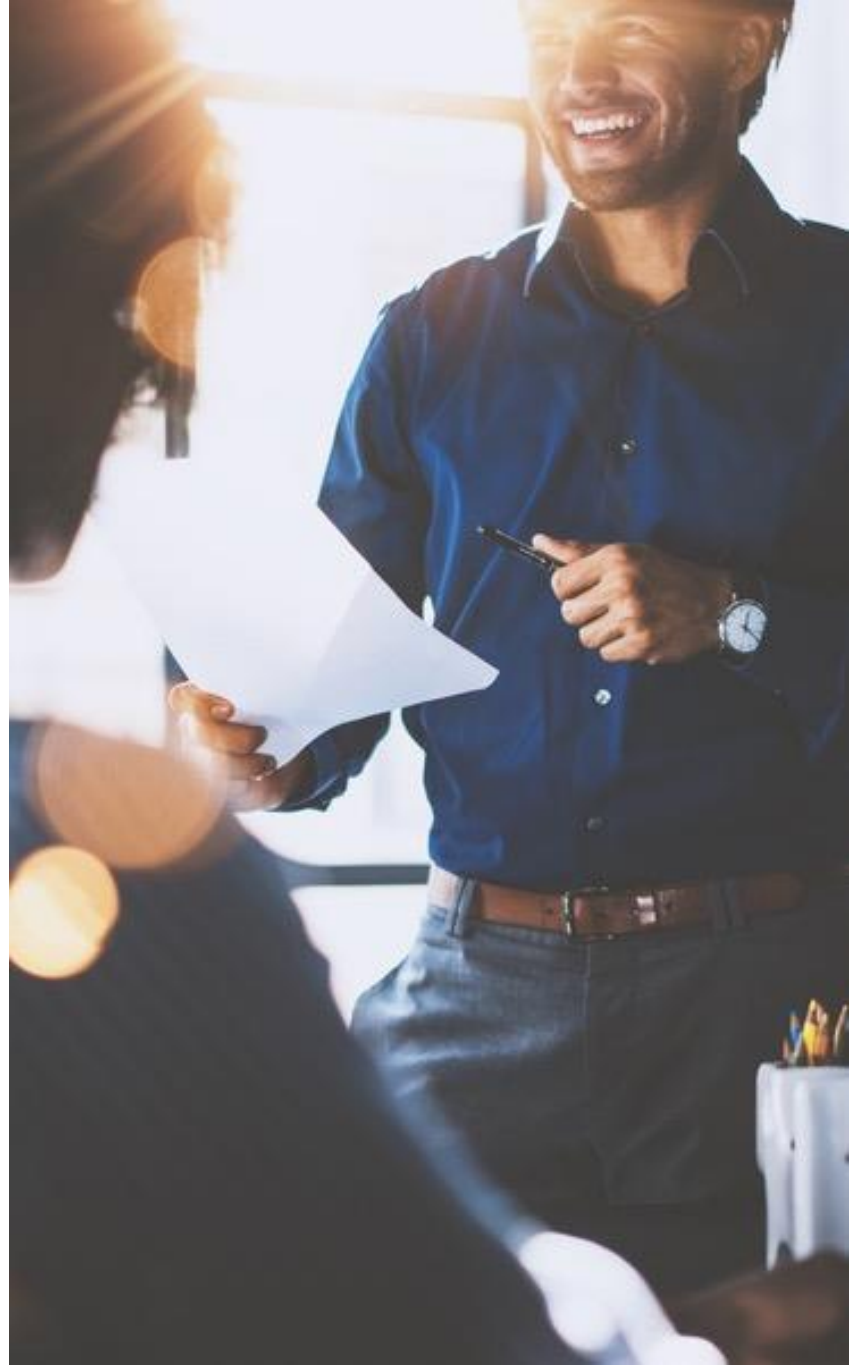


LEARNING TREE<sup>™</sup>  
INTERNATIONAL

# Objectives

---

- ▶ Describe the essentials of row functions
- ▶ Learn about mathematical functions
- ▶ Learn about string functions
- ▶ Learn about functions for date and time manipulation
- ▶ Use functions to convert between data types
- ▶ Use the CASE operator
- ▶ Use functions for handling NULL values





# Contents

---

## Row Function Fundamentals

- ▶ Mathematical Functions
- ▶ String Functions
- ▶ Date and Time Functions
- ▶ Data Type Conversion
- ▶ CASE
- ▶ NULL Functions



# Row Functions

- ▶ **Row functions are also called scalar functions**
  - Take input parameters and return a value
  - Repeatedly for each row processed by the statement
- ▶ **All database products have a large number of row functions**
  - Few of the functions are standardized
  - But many are similar across products

- ▶ **Example**

```
SELECT CompanyName, UPPER(CompanyName) AS UpperName
FROM   Shippers;
```

c7-01.sql

CompanyName	UpperName
Speedy Express	SPEEDY EXPRESS
United Package	UNITED PACKAGE
Federal Shipping	FEDERAL SHIPPING

# Contents

---

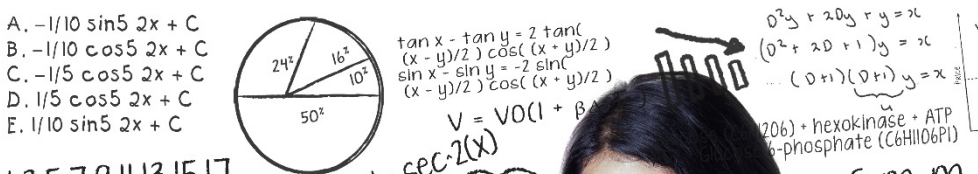
- ▶ Row Function Fundamentals

## Mathematical Functions

- ▶ String Functions
- ▶ Date and Time Functions
- ▶ Data Type Conversion
- ▶ CASE
- ▶ NULL Functions



# Mathematical Functions

- ▶ **Some mathematical functions are common among products even if not defined in the standard**
    - Square root: SQRT
    - Trigonometry: SIN, COS, etc.
  - ▶ **Some mathematical functions have similar functionality, but different names and parameter lists in different products**
    - Logarithms
    - Exponentiation
- 
- Handwritten notes and diagrams visible in the bottom right corner of the slide:
- A pie chart with a horizontal diameter. The top-left sector is labeled  $24^\circ$ , the top-right sector is labeled  $16^\circ$ , and the bottom sector is labeled  $50^\circ$ .
  - Trigonometric identities:
 
$$\tan x - \tan y = 2 \tan\left(\frac{x-y}{2}\right) \cos\left(\frac{x+y}{2}\right)$$

$$\sin x - \sin y = -2 \sin\left(\frac{x-y}{2}\right) \cos\left(\frac{x+y}{2}\right)$$
  - Algebraic equations:
 
$$0^2y + 20y + y = x$$

$$(0^2 + 20 + 1)y = x$$

$$\dots (0+1)(0+1)y = x$$
  - Other formulas:
 
$$V = VO(1 + \beta \dots)$$

$$\sec^2(x)$$

$$206 + \text{hexokinase} + \text{ATP} \rightarrow \text{6-phosphate (C6H10O6PI)}$$





# Rounding and Truncating

---

- ▶ **A common requirement is to round numeric values**
  - Particularly results of computations
- ▶ **ROUND (<value>,<scale>)**
  - The first parameter is the value or expression to be rounded
  - The second parameter is the scale
    - Can be positive to mean number of decimals
    - Zero means round to integer value (no decimals)
    - Can also be negative:
      - -1 means round to nearest 10
      - -2 means round to nearest hundred
      - -3 means round to nearest thousand
      - ...
      - -6 means round to nearest million
- ▶ **Truncating instead of rounding is product dependent**
  - PostgreSQL and Oracle: Use the TRUNC function
  - SQL Server: Add a third non-zero parameter to the ROUND function

# Rounding and Truncating Examples

- In Oracle, the second parameter can be omitted if zero

Expression	Result
ROUND(123.456,2)	123.46
TRUNC(123.456,2) -- PostgreSQL and Oracle ROUND(123.456,2,1) -- SQL Server	123.45
ROUND(456.789,0)	457
TRUNC(456.789,0) -- PostgreSQL and Oracle ROUND(456.789,0,1) -- SQL Server	456
ROUND(456,-2)	500
TRUNC(456,-2) -- PostgreSQL and Oracle ROUND(456,-2,1) -- SQL Server	400

# Rounding and Truncating Example

## ► Example: Round or truncate to integer (zero decimals)

c7-02.sql

The first parameter is the value to be rounded

The second parameter is the number of decimals

```
SELECT UnitPrice
       ,ROUND(UnitPrice,0) AS RoundPrice
       ,TRUNC(UnitPrice,0) AS TruncPrice    -- PostgreSQL and Oracle
       ,ROUND(UnitPrice,0,1) AS TruncPrice -- SQL Server
FROM   Products
WHERE  ProductId BETWEEN 25 AND 30
ORDER BY UnitPrice;
```

Notice that SQL Server uses a different function for truncating

UnitPrice	RoundPrice	TruncPrice
14.00	14.00	14.00
25.89	26.00	25.00
31.23	31.00	31.00
43.90	44.00	43.00
45.60	46.00	45.00
123.79	124.00	123.00

# Rounding Example

## ► Example: Round to nearest thousand

c7-03.sql

```
SELECT CurrentSalary
       ,ROUND(CurrentSalary,-3) AS Thousands
FROM   Employees;
```

CurrentSalary	Thousands
-----	-----
2000	2000
3000	3000
3000	3000
<b>3600</b>	<b>4000</b>
4000	4000
5000	5000
<b>5200</b>	<b>5000</b>
6000	6000
9000	9000

–3 means round to the  
nearest thousand

# Contents

---

- ▶ Row Function Fundamentals
- ▶ Mathematical Functions

## String Functions

- ▶ Date and Time Functions
- ▶ Data Type Conversion
- ▶ CASE
- ▶ NULL Functions



# UPPER and LOWER

- ▶ We have already seen an example of the UPPER function that returns all uppercase letters
- ▶ There is also a function LOWER that returns all lowercase letters
- ▶ PostgreSQL and Oracle also have a function INITCAP that capitalizes the first letter of each word
  - Example

c7-04.sql

```
SELECT CompanyName, INITCAP(CompanyName) AS Initcapped
FROM Customers
WHERE CompanyName LIKE 'F%';
```

COMPANYNAME	INITCAPPED
FISSA Fabrica Inter. Salchichas S.A.	Fissa Fabrica Inter. Salchichas S.A.
Familia Arquibaldo	Familia Arquibaldo
Folies gourmandes	Folies <b>G</b> ourmandes
Folk och fä HB	Folk <b>O</b> ch <b>F</b> ä <b>H</b> b
France restauration	France <b>R</b> estaurat <b>i</b> on
Franchi S.p.A.	Franchi <b>S.P.A.</b>
Frankenversand	Frankenversand
Furia Bacalhau e Frutos do Mar	Furia Bacalhau <b>E</b> Frutos <b>D</b> o Mar

# SUBSTRING

---

- ▶ **SUBSTRING extracts part of a character string, from a starting point up to a specified length**
  - Oracle and SQL Server both deviate from the standard
- ▶ **Standard: SUBSTRING (<string> FROM <start> [FOR <length>])**
  - If the FOR <length> clause is omitted, it goes to the end of the string
- ▶ **Oracle: SUBSTR (<string>, <start> [,<length>])**
  - The function name is SUBSTR
  - If the <length> parameter is omitted, it goes to the end of the string
- ▶ **SQL Server: SUBSTRING (<string>, <start>, <length>)**
  - The <length> parameter may *not* be omitted
- ▶ **PostgreSQL accepts all three varieties**



# SUBSTRING Example

c7-05.sql

The first parameter is  
the string to extract from

The third parameter  
is the length

```
SELECT CompanyName  
       ,SUBSTRING(CompanyName FROM 4 FOR 8) AS NameSub  
FROM   Shippers;
```

The second parameter is  
the start position

CompanyName	NameSub
Speedy Express	edy Expr
United Package	ted Pack
Federal Shipping	eral Shi

# Trimming

---

- ▶ **The standard specifies a function TRIM, which removes characters from the beginning or end of a string**
- ▶ **Syntax: TRIM([<side>] [<character>] FROM <string>)**
- ▶ **The <side> parameter can be one of the keywords LEADING, TRAILING, or BOTH**
  - BOTH is default if the parameter is omitted
- ▶ **The <character> parameter specifies which character to remove**
  - Can be one character only
  - Space is default if the parameter is omitted
- ▶ **PostgreSQL has a slightly different syntax**
  - TRIM([<side>] FROM <string>), [<characters>])
  - Can trim multiple characters
  - Space is default if the parameter is omitted

# Product Specifics

---

- ▶ **SQL Server does not have the TRIM function**
  - But it does have the two functions LTRIM and RTRIM
  - Remove leading (L means LEFT) or trailing (R means RIGHT) spaces
  - Only spaces can be trimmed
    - Not allowed to specify other characters
  - To trim from both sides, use both functions
    - The functionality of TRIM(BOTH ' ' FROM <string>) is achieved with LTRIM(RTRIM(<string>))
- ▶ **PostgreSQL and Oracle have the TRIM function as well as LTRIM and RTRIM**
  - LTRIM and RTRIM supports trimming multiple characters
    - This is rarely used
- ▶ **PostgreSQL also has the BTRIM function as well as LTRIM and RTRIM**
  - Trims from both sides and supports trimming multiple characters
    - This is rarely used

# Contents

---

- ▶ Row Function Fundamentals
- ▶ Mathematical Functions
- ▶ String Functions

## Date and Time Functions

- ▶ Data Type Conversion
- ▶ CASE
- ▶ NULL Functions



# Date and Time Functions

---

- ▶ **The area of date and time handling is poorly standardized, and there are huge product differences**
  - This applies to data types as well as to functions for handling them
- ▶ **The standard specifies functions for returning date and time from the machine clock**
  - CURRENT\_DATE returns the date
    - Oracle supports the function, but deviates from the standard by including time of day, like a timestamp
    - Same as Oracle's proprietary SYSDATE
    - SQL Server does not support the function
  - CURRENT\_TIME returns the time of day
    - Not supported by Oracle and SQL Server
  - CURRENT\_TIMESTAMP returns both date and time
    - Similar to Oracle's SYSTIMESTAMP
    - Same as SQL Server's GETDATE()

# Extracting Date and Time Parts

- ▶ **The standard specifies a function EXTRACT that returns a specific part of a datetime expression**

- Syntax: `EXTRACT(<part> FROM <expression>)`

- ▶ **Example**

```
SELECT BirthDate
       ,EXTRACT(YEAR FROM BirthDate) AS BirthYear
       ,EXTRACT(MONTH FROM BirthDate) AS BirthMonth
       ,EXTRACT(DAY FROM BirthDate) AS BirthDay
FROM   Employees;
```

c7-06.sql

This example  
works in Oracle

BIRTHDATE	BIRTHYEAR	BIRTHMONTH	BIRTHDAY
-----	-----	-----	-----
19-FEB-1952	1952	2	19
04-MAR-1955	1955	3	4
08-DEC-1948	1948	12	8
30-AUG-1963	1963	8	30
19-SEP-1937	1937	9	19
02-JUL-1963	1963	7	2
29-MAY-1960	1960	5	29
09-JAN-1958	1958	1	9
27-JAN-1966	1966	1	27

# Extracting Date and Time Parts

- ▶ SQL Server does *not* support the EXTRACT function
- ▶ In SQL Server, a similar result can be achieved by using the YEAR, MONTH, and DAY functions

```
SELECT BirthDate
       ,YEAR(BirthDate) AS BirthYear
       ,MONTH(BirthDate) AS BirthMonth
       ,DAY(BirthDate) AS BirthDay
FROM   Employees;
```

c7-06.sql

This example works  
in SQL Server

- ▶ **Product-specific alternatives:**
  - SQL Server has the DATEPART and DATENAME functions, plus the <style> parameter of the CONVERT function
  - PostgreSQL and Oracle have the <format> parameter of the TO\_CHAR function
  - The product specifics are complex and are covered in detail in the product-specific courses



# Date and Time Calculation

---

- ▶ **Calculations on dates and times are very product dependent**
- ▶ **Adding or subtracting time intervals**
  - Example shown on next slide
  - Oracle can simply add or subtract a number of days
  - Works for the DATETIME data type in SQL Server, and DATE data type in PostgreSQL
- ▶ **Adding or subtracting other date and time parts**
  - Oracle: ADD\_MONTHS function for handling months intelligently
  - Oracle: INTERVAL handles all date components
    - But months may fail because of varying length
  - SQL Server: DATEADD handles all date components
- ▶ **Calculating intervals**
  - PostgreSQL and Oracle: Simply subtracting one DATETIME from another gives the interval in days, including fractions
  - SQL Server: DATEDIFF gives integer number of specific component

# Date Calculation Example

- Assume that orders that have been shipped will be invoiced with a due date of 30 days after shipping

c7-07.sql

```
SELECT OrderID
       ,ShippedDate
       ,ShippedDate + interval '30 days' AS DueDate
FROM   Orders
WHERE  ShippedDate IS NOT NULL
ORDER BY OrderId;
```

OrderID	ShippedDate	DueDate
10248	1996-07-16	1996-08-15
10249	1996-07-10	1996-08-09
10250	1996-07-12	1996-08-11
10251	1996-07-15	1996-08-14
10252	1996-07-11	1996-08-10
10253	1996-07-16	1996-08-15
.....		

# Contents

---

- ▶ Row Function Fundamentals
- ▶ Mathematical Functions
- ▶ String Functions
- ▶ Date and Time Functions

## Data Type Conversion

- ▶ CASE
- ▶ NULL Functions



# Data Type Conversion

- ▶ **All products perform implicit conversion between data types when required**
  - However, implicit conversion can have tricky side effects
    - So it is recommended to control the conversion explicitly using data type conversion functions
- ▶ **The standard specifies a function CAST for data type conversion**
  - Syntax: CAST(<value> AS <datatype>)

- ▶ **Example**

```
SELECT CAST(CurrentSalary as CHAR(7)) AS CharSal  
FROM Employees;
```

```
CharSal  
-----  
2000.00  
3000.00  
3000.00  
3600.00  
.....
```

c7-09.sql

# Product-Specific Data Type Conversion

---

- ▶ **PostgreSQL, Oracle, and SQL Server support the CAST function**
  - But they also have product-specific functions that allow more fine-grained control
- ▶ **PostgreSQL and Oracle**
  - TO\_NUMBER, TO\_CHAR, TO\_DATE, and TO\_TIMESTAMP
  - Convert into respective data types
  - Use format masks for exact format specification
- ▶ **SQL Server**
  - CONVERT uses a style parameter for format specification
  - STR converts from numeric values to character

## Hands-On Exercise 7.1

**In your Exercise Manual, please refer to Hands-On Exercise 7.1: Math and String Functions**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**





# Contents

---

- ▶ Row Function Fundamentals
- ▶ Mathematical Functions
- ▶ String Functions
- ▶ Date and Time Functions
- ▶ Data Type Conversion

## CASE

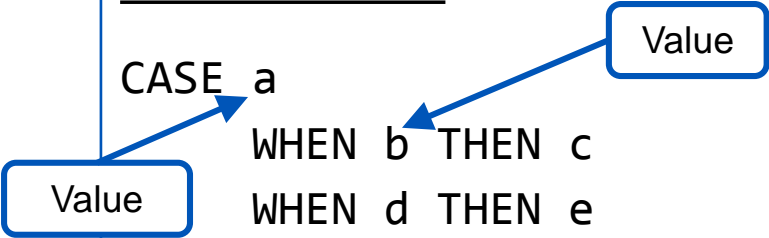
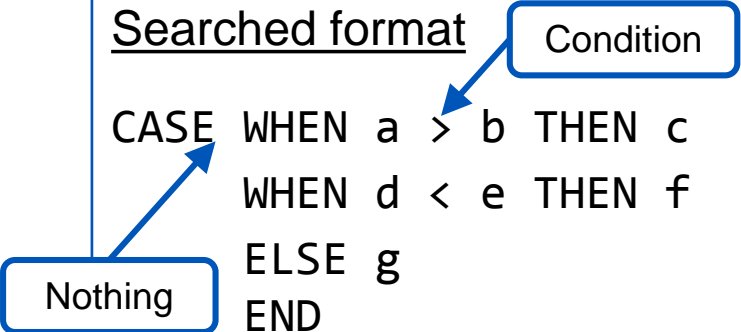
- ▶ NULL Functions





# CASE

- ▶ The CASE operator is SQL's way of performing IF tests
- ▶ CASE has two different formats, *selector* and *searched*
  - The two formats may not be mixed

<u>Selector format</u>		<u>Meaning</u>
 <pre>CASE a   WHEN b THEN c   WHEN d THEN e   ELSE f END</pre>		IF a = b THEN RETURN c ELSE IF a = d THEN RETURN e ELSE RETURN f
<u>Searched format</u>		<u>Meaning</u>
 <pre>CASE WHEN a &gt; b THEN c       WHEN d &lt; e THEN f       ELSE g       END</pre>		IF a > b THEN RETURN c ELSE IF d < e THEN RETURN f ELSE RETURN g

# CASE

---

- ▶ **Note that the expression starts with the keyword CASE and ends with the keyword END**
- ▶ **Processing stops when the first true condition is encountered**
- ▶ **If the ELSE clause is omitted, NULL will be returned if none of the test conditions are true**
- ▶ **Return data types**
  - Oracle requires all return values to have the same data type
  - SQL Server performs implicit data type conversion when required
  - Conversion is not performed if the processing has stopped on an earlier test
    - Consequently, conversion problems may not necessarily be uncovered
- ▶ **Recommendations:**
  - Use conversion functions when required to avoid implicit conversion
  - Make sure your test data covers all the test conditions

# Selector CASE Example

## ► Replace the gender abbreviation with the full word

c7-11.sql

```
SELECT EmployeeID, FirstName, LastName
       ,CASE Gender
           WHEN 'F' THEN 'Female'
           WHEN 'M' THEN 'Male'
       END AS Gender
FROM   Employees;
```

Column alias

EmployeeID	FirstName	LastName	Gender
1	Nancy	Davolio	Female
2	Andrew	Fuller	Male
3	Janet	Leverling	Female
4	Margaret	Peacock	Female
5	Steven	Buchanan	Male
6	Michael	Suyama	Male
7	Robert	King	Male
8	Laura	Callahan	Female
9	Anne	Dodsworth	Female

# Searched CASE Example

- Range the products into low, medium, and high price ranges

c7-12.sql

```
SELECT ProductID
       ,ProductName
       ,UnitPrice
       ,CASE WHEN UnitPrice < 10 THEN 'Low'
             WHEN UnitPrice >= 100 THEN 'High'
             ELSE 'Medium'
       END AS PriceRange
FROM   Products;
```

ProductID	ProductName	UnitPrice	PriceRange
1	Chai	18	Medium
13	Konbu	6	Low
14	Tofu	23.25	Medium
29	Thüringer Rostbratwurst	123.79	High
30	Nord-Ost Matjeshering	25.89	Medium
41	Jack's New England Clam Chowder	9.65	Low
.....			

# Contents

---

- ▶ Row Function Fundamentals
- ▶ Mathematical Functions
- ▶ String Functions
- ▶ Date and Time Functions
- ▶ Data Type Conversion
- ▶ CASE

## NULL Functions



# Functions for Handling NULL Values

---

- ▶ **A NULL value in an expression causes the result of the expression to be NULL**
  - Remember the example of CurrentSalary + Commission
- ▶ **The standard COALESCE function can be used to treat NULL values as some real value**
  - Takes a number of parameters and returns the first non-NULL parameter value
  - Returns NULL if all parameter values are NULL
- ▶ **Syntax: COALESCE(<param1>, <param2>, ..., <paramn>)**
  - Meaning:

```
CASE WHEN <param1> IS NOT NULL THEN <param1>
      WHEN <param2> IS NOT NULL THEN <param2>
      ...
      WHEN <paramn> IS NOT NULL THEN <paramn>
      ELSE NULL
END
```

# COALESCE Examples

- Assume that those who have no defined commission will receive a commission of 50

c7-13.sql

```
SELECT FirstName
       ,LastName
       ,CurrentSalary
       ,COALESCE(Commission,50) AS Commission
       ,CurrentSalary+COALESCE(Commission,50) AS TotalPay
FROM   Employees;
```

FirstName	LastName	CurrentSalary	Commission	TotalPay
-----	-----	-----	-----	-----
Nancy	Davolio	2000	50	2050
Andrew	Fuller	9000	50	9050
Janet	Leverling	6000	50	6050
Margaret	Peacock	3000	50	3050
Steven	Buchanan	5000	50	5050
<b>Michael</b>	<b>Suyama</b>	<b>3600</b>	<b>200</b>	<b>3800</b>
<b>Robert</b>	<b>King</b>	<b>3000</b>	<b>400</b>	<b>3400</b>
Laura	Callahan	4000	50	4050
<b>Anne</b>	<b>Dodsworth</b>	<b>5200</b>	<b>300</b>	<b>5500</b>



# COALESCE Examples

- Assume that those who have no defined commission will receive a commission that is 10 percent of their salary

c7-14.sql

```
SELECT FirstName
       ,LastName
       ,CurrentSalary
       ,COALESCE(Commission,CurrentSalary/10) AS Commission
       ,CurrentSalary+COALESCE(Commission,CurrentSalary/10)
                                     AS TotalPay
FROM   Employees;
```

FirstName	LastName	CurrentSalary	Commission	TotalPay
Nancy	Davolio	2000	200	2200
Andrew	Fuller	9000	900	9900
Janet	Leverling	6000	600	6600
Margaret	Peacock	3000	300	3300
Steven	Buchanan	5000	500	5500
Michael	Suyama	3600	200	3800
Robert	King	3000	400	3400
Laura	Callahan	4000	400	4400
Anne	Dodsworth	5200	300	5500

# Product-Specific Functions

---

- ▶ **PostgreSQL, Oracle, and SQL Server support the COALESCE function**
- ▶ **Oracle and SQL Server both have product-specific functions with similar functionality, but limited to two parameters only**
  - Oracle: NVL(<param1>, <param2>)
  - SQL Server: ISNULL(<param1>, <param2>)

## Hands-On Exercise 7.2

**In your Exercise Manual, please refer to Hands-On Exercise 7.2: NULL Functions and CASE**

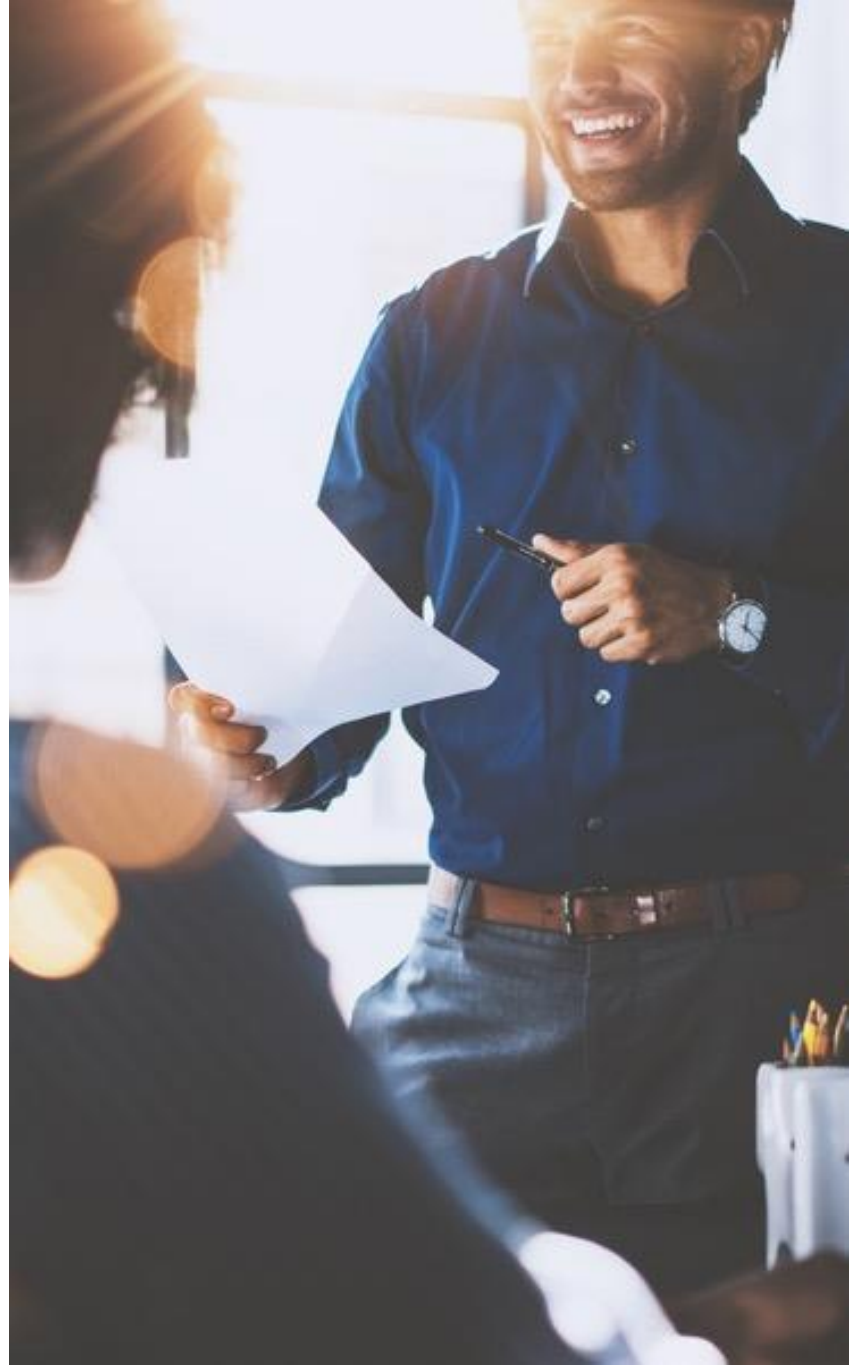
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Describe the essentials of row functions
- ▶ Learn about mathematical functions
- ▶ Learn about string functions
- ▶ Learn about functions for date and time manipulation
- ▶ Use functions to convert between data types
- ▶ Use the CASE operator
- ▶ Use functions for handling NULL values



# Chapter 8

## Aggregate Functions



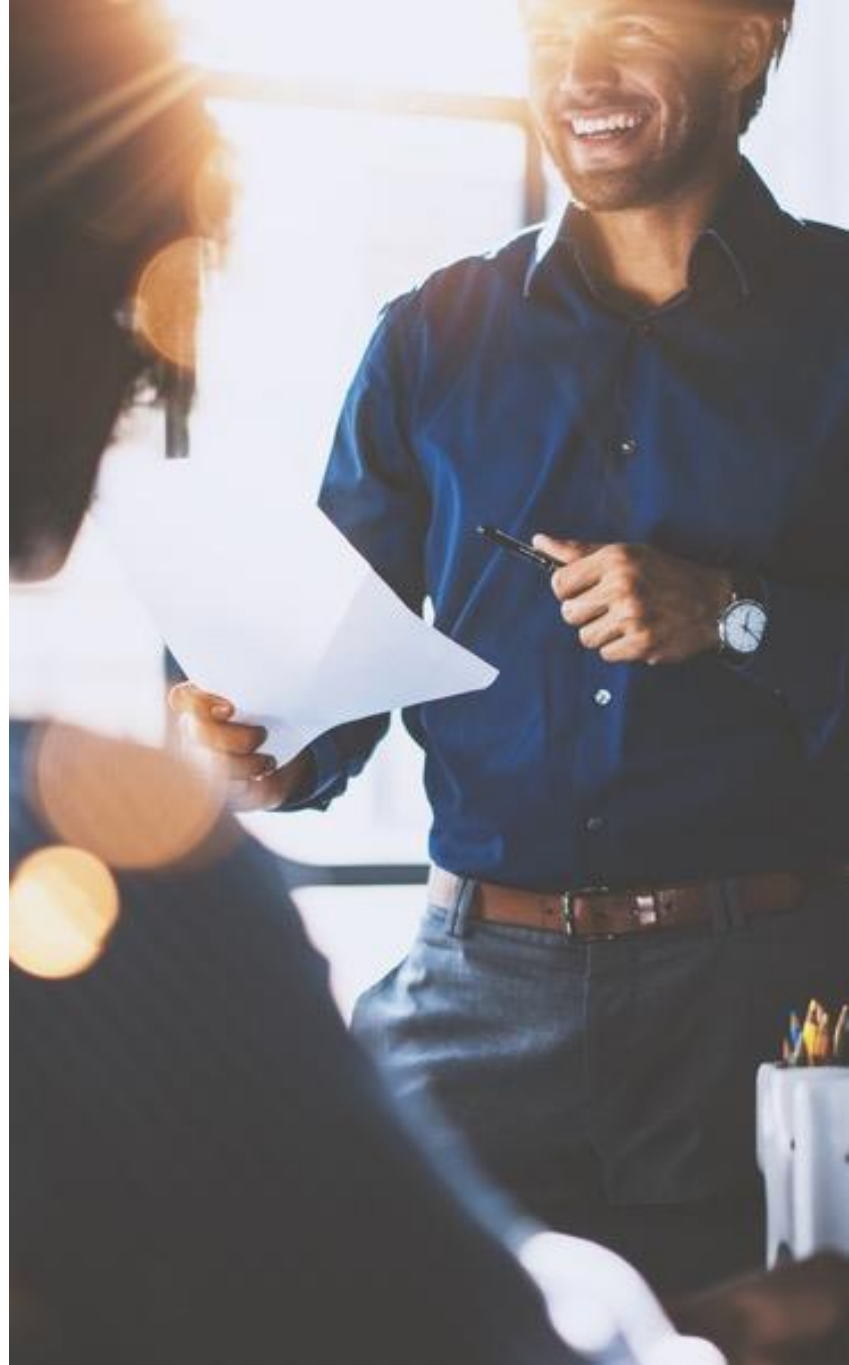
LEARNING TREE™  
INTERNATIONAL



# Objectives

---

- ▶ Use aggregate functions to produce statistics reports
- ▶ Use GROUP BY
- ▶ Apply the HAVING filter



# Contents

---

## Aggregate Functions

- ▶ GROUP BY
- ▶ HAVING





# Aggregate Functions

- ▶ An aggregate function views a number of rows as a group and returns an aggregated value as one row

- ▶ The standard aggregate functions are

SUM    AVG    MAX    MIN    COUNT

- ▶ Example of aggregate functions

c8-01.sql

```
SELECT SUM(CurrentSalary) AS SumSal
      ,AVG(CurrentSalary) AS AvgSal
      ,ROUND(AVG(CurrentSalary),0) AS AvgRounded
      ,MIN(CurrentSalary) AS MinSal
      ,MAX(CurrentSalary) AS MaxSal
FROM   Employees;
```

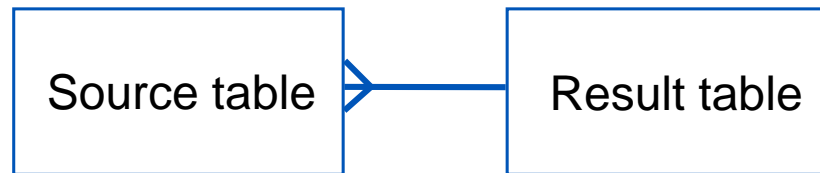
Suppress  
the  
decimals

SumSal	AvgSal	AvgRounded	MinSal	MaxSal
40800	4533.3333	4533	2000	9000

# Grouping and Aggregates

---

- ▶ **Whenever an aggregate function or the GROUP BY keyword is used, the structure of the query will change**
- ▶ **There will now be a many-to-one relationship between the rows in the source table and the result table**



- The source table is the table from which the query fetches values
- The source table may be a real table or a view, or a join of several tables
- The result table is the result of the query
- The result table (also called result set) is not a real table, but it has the structure of a table (rows and columns)

# The COUNT Function

---

## ► The COUNT function takes three types of arguments

1. COUNT(\*)
  - Returns the number of rows
2. COUNT(<column>)
  - Returns the number of rows in which the specified column has a value
  - NULL values are not included
3. COUNT(DISTINCT <column>)
  - Returns the number of different values in the specified column
  - NULL values are *not* included

# COUNT Examples

## ► How many employees do we have?

```
SELECT COUNT(*) AS Emps
FROM   Employees;
```

- There are nine rows in the table

```
Emps
----
9
```

## ► How many employees have a manager?

```
SELECT COUNT(ReportsTo) AS HasMgr
FROM   Employees;
```

- Eight of the rows have a value in the ReportsTo column

```
HasMgr
-----
8
```

## ► How many managers do we have?

```
SELECT COUNT(DISTINCT ReportsTo) AS IsMgr
FROM   Employees;
```

- There are two different values in the ReportsTo column, NULL values not included

```
IsMgr
-----
2
```

c8-02.sql

```
SELECT ReportsTo
FROM   Employees
```

```
ReportsTo
-----
```

```
2
NULL
2
2
2
5
5
2
5
```

# Aggregate Functions and NULL

## ► All aggregate functions ignore NULL values

- Example:

c8-03.sql

```
SELECT COUNT(Commission) AS Count
       ,AVG(Commission)   AS Avg
       ,SUM(Commission)   AS Sum
       ,MAX(Commission)   AS Max
       ,MIN(Commission)   AS Min
FROM   Employees;
```

Count	Avg	Sum	Max	Min
3	300	900	400	200

```
SELECT Commission
FROM   Employees
Commission
-----
NULL
NULL
NULL
NULL
NULL
200
400
NULL
300
```

# Complex Expression With Aggregate Function

## ► Parameters to aggregate functions can be complex expressions

c8-04.sql

- Can include scalar functions
- Example:

```
SELECT SUM(CurrentSalary+COALESCE(Commission,0)) AS SumSal  
FROM Employees;
```

SumSal

-----

41700

The parameter can be an expression

## Hands-On Exercise 8.1

**In your Exercise Manual, please refer to  
Hands-On Exercise 8.1: Aggregate Functions**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**





# Contents

---

- ▶ **Aggregate Functions**

- GROUP BY**

- ▶ **HAVING**



# GROUP BY

---

- ▶ **GROUP BY groups the source table rows so that the result table will have one row per group**
  - Aggregate functions are then computed within each group
- ▶ **If the query has *aggregate functions* and no GROUP BY, the result will *always* have exactly one row**
  - This is true even if the source table is empty or if no rows in the table satisfy the WHERE condition
- ▶ **With GROUP BY, the result may have any number of rows (zero, one, or several)**
  - Determined by the number of different values in the GROUP BY columns

# GROUP BY Examples

- Find the number of products per category

c8-05.sql

```
SELECT CategoryID
       ,COUNT(*) AS Products
FROM   Products
GROUP BY CategoryID
ORDER BY CategoryID;
```

CategoryID	Products
-----	-----
1	12
2	12
3	13
4	10
5	7
6	6
7	5
8	13

# GROUP BY Examples

- Find the number of employees and the total salary per division

c8-06.sql

```
SELECT DivisionID
        ,COUNT(*) AS Employees
        ,SUM(CurrentSalary) AS SumSal
FROM    Employees
GROUP BY DivisionID
ORDER BY DivisionID
```

DivisionID	Employees	SumSal
1	4	22000
2	5	18800

# GROUP BY on Multiple Columns

- ▶ It is perfectly possible to group on more than one column
- ▶ Find the number of employees and the total salary per *department*
  - Each division consists of several departments
  - Departments are identified by two columns, DivisionID and DepartmentID

```
SELECT DivisionID
       ,DepartmentID
       ,COUNT(*) AS Employees
       ,SUM(CurrentSalary) AS SumSal
FROM   Employees
GROUP BY DivisionID, DepartmentID
ORDER BY DivisionID, DepartmentID;
```

c8-07.sql

Columns separated  
by comma

DivisionID	DepartmentID	Employees	SumSal
1	100	2	11000
1	200	2	11000
2	200	1	3000
2	300	4	15800

# Contents

---

- ▶ **Aggregate Functions**
- ▶ **GROUP BY**

## **HAVING**



# Grouping and Filtering

---

## ► WHERE clause

- Filters from the source table
- Determines which rows to include in the groups
- Is executed *before* the grouping takes place and aggregates are computed



### **Warning:**

**The WHERE clause tests single-row values**

**The WHERE clause cannot test aggregate values**

---

## ► HAVING clause

- Filters from the result of the aggregation
- Determines which result groups are to be presented in the final result
- Is executed *after* the grouping has taken place and aggregates computed



### **Warning:**

**The HAVING clause tests aggregate values**

**The HAVING clause cannot test single-row values**

---



# HAVING Examples

Do Now

- **Modify example c8-05.sql on Slide 8-13 to show only categories with 10 products or more**

c8-08.sql

```
SELECT CategoryID
       ,COUNT(*) AS Products
FROM   Products
GROUP BY CategoryID
HAVING COUNT(*) >= 10
ORDER BY CategoryID;
```

CategoryID	Products
-----	-----
1	12
2	12
3	13
4	10
8	13

# HAVING Examples

Do Now

- **Modify example c8-07.sql on Slide 8-15 to exclude departments with only one employee**

c8-09.sql

```
SELECT DivisionID
       ,DepartmentID
       ,COUNT(*) AS Employees
       ,SUM(CurrentSalary) AS SumSal
FROM   Employees
GROUP BY DivisionID, DepartmentID
HAVING COUNT(*) > 1
ORDER BY DivisionID, DepartmentID;
```

DivisionID	DepartmentID	Employees	SumSal
1	100	2	11000
1	200	2	11000
2	300	4	15800

## Hands-On Exercise 8.2

In your Exercise Manual, please refer to  
**Hands-On Exercise 8.2: GROUP BY and HAVING**

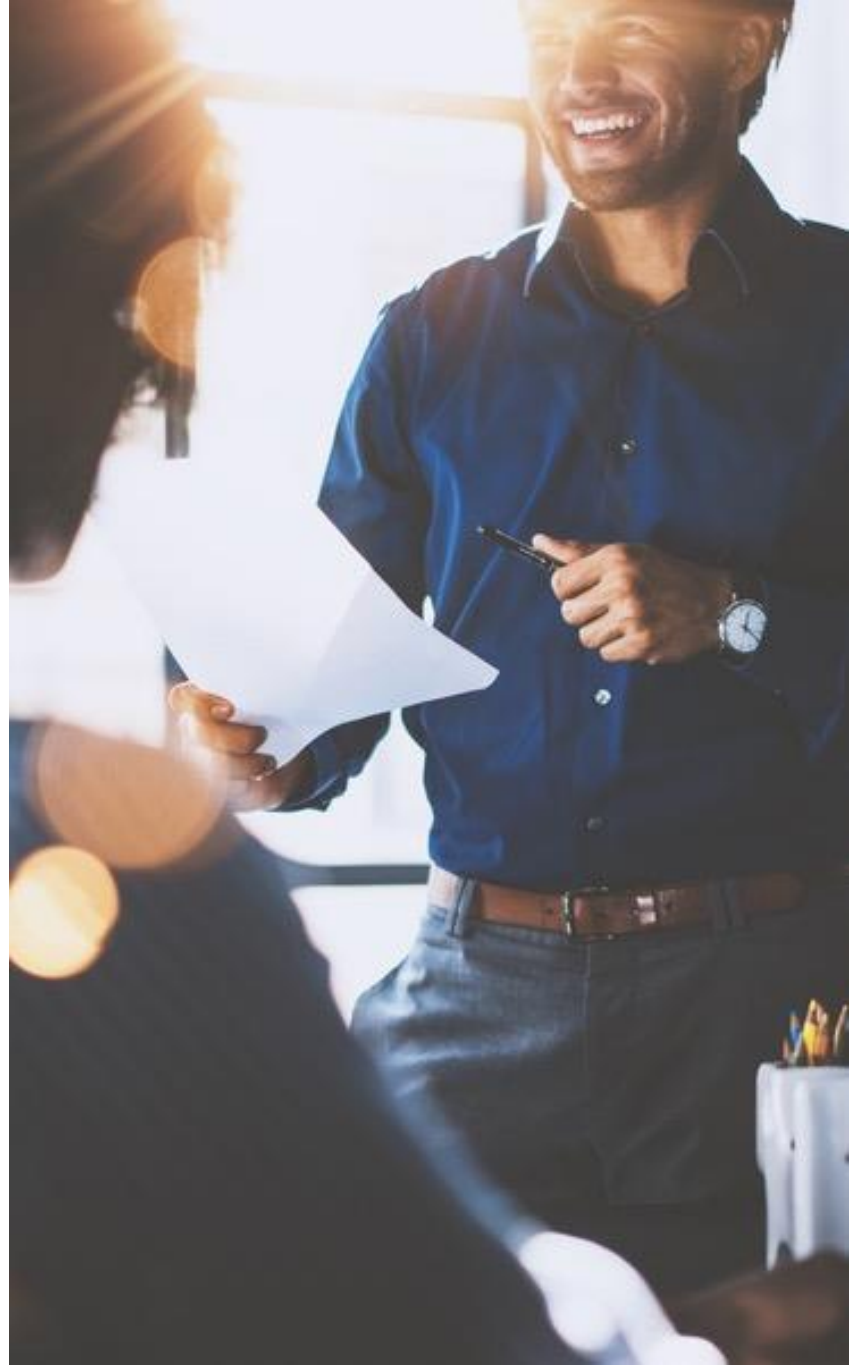
- ▶ **Solutions to all the exercises are provided as files located in the folder C:\Course925Solutions**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Use aggregate functions to produce statistics reports
- ▶ Use GROUP BY
- ▶ Apply the HAVING filter



# Chapter 9

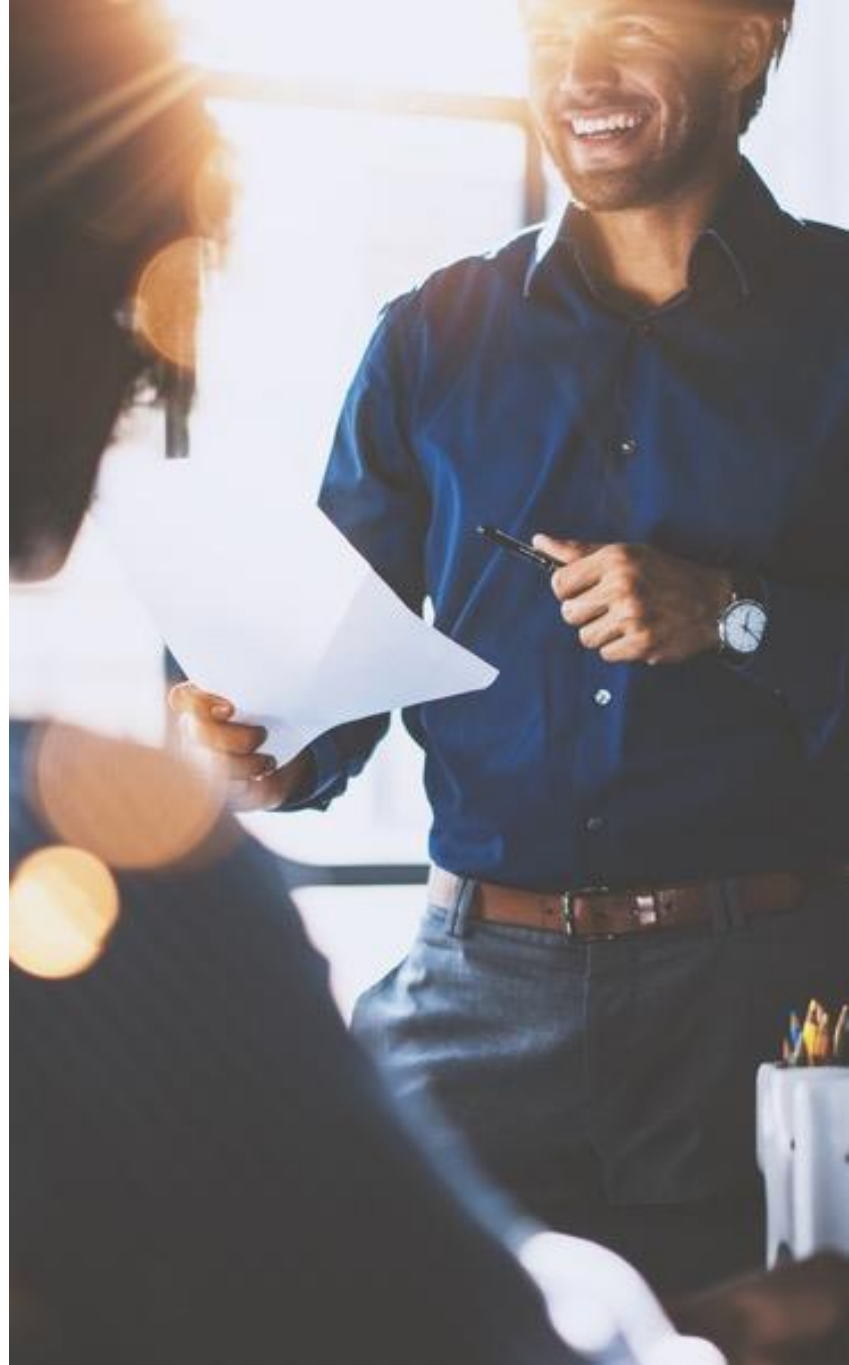
## Subqueries



# Objectives

---

- ▶ Learn the difference between simple and correlated subqueries
- ▶ Use subqueries in conditions and expressions
- ▶ Use the EXISTS operator with subqueries



# Contents

---

## Subqueries

- ▶ Correlated Subqueries
- ▶ EXISTS





# Subqueries

---

- ▶ **A SELECT statement embedded into another statement is called a *subquery***
  - The syntax requires that a subquery is enclosed in parentheses
- ▶ **There are different ways of categorizing subqueries**
  - Correlated (also called *synchronized*) or non-correlated (also called *unsynchronized, simple, or self-contained*)
  - Single-row or multi-row
  - Single-column or multi-column
- ▶ **Terminology**
  - Single-row/single-column subqueries are also called scalar subqueries
    - Returns exactly one value
  - Single-row/multi-column subqueries are also called row subqueries
  - Multi-row subqueries are also called table subqueries
    - Regardless of single-column or multi-column

# Subquery Example

## ► Find the most expensive product

c9-01.sql

```
SELECT ProductID
       ,ProductName
       ,UnitPrice
FROM   Products
WHERE  UnitPrice = (SELECT MAX(UnitPrice) FROM Products);
```

The subquery  
returns one value

ProductID	ProductName	UnitPrice
38	Côte de Blaye	263.5

- The subquery finds the maximum unit price: 263.5
- The WHERE clause in the main query finds the product that has that price

# Multirow Subquery

- ▶ **In the previous example, only single-row subqueries are allowed**
  - The contexts require a single value
- ▶ **Multirow subqueries are allowed in contexts where multiple values are meaningful**
  - For example, with the IN and NOT IN operators
- ▶ **Example: Find the conference rooms that have no reservations**
  - That is, rooms that have no matching rows in the Reservations table

```
SELECT RoomID, Name
FROM   ConferenceRooms
WHERE  RoomID NOT IN (SELECT RoomID FROM Reservations);
```

c9-02.sql

RoomID	Name
-----	-----
225	2nd Floor Small room
310	3rd Floor Tiny room

The subquery returns multiple values

# NOT IN and NULL

- ▶ **Beware of NULL values in subqueries used with NOT IN**
  - If there are NULL values in the subquery result, then the NOT IN condition will not be satisfied
- ▶ **Example: Find the employees who do not have anybody reporting to them**
  - Should be all except 2 and 5

```
SELECT EmployeeID
       ,LastName
FROM   Employees
WHERE  EmployeeID NOT IN
      (SELECT ReportsTo FROM Employees);
```

(0 row(s) affected)

The subquery  
result contains a  
NULL value

ReportsTo
-----
2
NULL
2
2
2
5
5
2
5

c9-03.sql

# NOT IN and NULL

## ► Exclude NULL values to achieve the correct result

c9-04.sql

```
SELECT EmployeeID
       ,LastName
FROM   Employees
WHERE  EmployeeID NOT IN
      (SELECT ReportsTo FROM Employees
       WHERE ReportsTo IS NOT NULL);
```

EmployeeID	LastName
1	Davolio
3	Leverling
4	Peacock
6	Suyama
7	King
8	Callahan
9	Dodsworth

Exclude the NULL from  
the subquery result

(7 row(s) affected)

# Subqueries in Expressions

- ▶ **Example: Show the employees with a salary below average, including the difference from the average**
  - This example shows how subqueries can help overcome SQL limitations

- ▶ **The following is *not* permitted**

```
SELECT LastName, CurrentSalary
      ,AVG(CurrentSalary) AS Average
      ,CurrentSalary-AVG(CurrentSalary) AS Difference
FROM   Employees
WHERE  CurrentSalary<AVG(CurrentSalary);
```

Illegal

Illegal

Illegal

c9-05.sql

- Mix of row values and aggregate values is not allowed
  - Aggregates may not be used in the WHERE clause
- ▶ **Solution: Embed the aggregate calculations in subqueries**

# Subqueries in Expressions

- The subqueries are executed first and return a scalar value that may be referenced in the main query

c9-06.sql

```
SELECT LastName, CurrentSalary
      ,(SELECT AVG(CurrentSalary) FROM Employees) AS Average
      ,CurrentSalary-(SELECT AVG(CurrentSalary) FROM Employees)
                        AS Difference
FROM   Employees
WHERE  CurrentSalary<(SELECT AVG(CurrentSalary) FROM Employees);
```

LastName	CurrentSalary	Average	Difference
Davolio	2000	4533.3333	-2533.3333
Peacock	3000	4533.3333	-1533.3333
King	3000	4533.3333	-1533.3333
Suyama	3600	4533.3333	-933.3333
Callahan	4000	4533.3333	-533.3333

Each subquery  
returns a value



## Hands-On Exercise 9.1

**In your Exercise Manual, please refer to Hands-On Exercise 9.1: Subqueries**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**

# Contents

---

- ▶ Subqueries

## Correlated Subqueries

- ▶ EXISTS



# Correlated Subqueries

---

- ▶ **The subqueries we have seen so far have been able to execute independently of the main query**
  - The subquery executes first, and the result of the subquery is then used in the main query
  - Such a subquery is called *non-correlated*, *unsynchronized*, *self-contained*, or *simple*
- ▶ **A subquery may reference columns from the main query table(s)**
  - Such a subquery is *not* able to execute on its own
    - Data values from the main query table(s) are required
    - The subquery can “see” the main query table(s)
  - Such a subquery is called *correlated* or *synchronized*
- ▶ **Data values from the main query table(s) will be different for the different rows processed by the main query**
  - Therefore, a correlated subquery is conceptually executed repeatedly for each of the main query rows
  - Each execution uses a different value from the main query table(s)

# Correlated Subquery Example

Do Now

- ▶ The query can be modified below to find the most expensive product within each category rather than the overall most expensive
  - Correlate the subquery on CategoryID

```
SELECT ProductID
       ,ProductName
       ,UnitPrice
FROM   Products
WHERE  UnitPrice = (SELECT MAX(UnitPrice) FROM Products);
```

c9-01.sql

ProductID	ProductName	UnitPrice
38	Côte de Blaye	263.5

# Correlated Subquery Example

Do Now

## ► Modified query

```
SELECT CategoryID, ProductID, ProductName, UnitPrice
FROM   Products p1
WHERE  UnitPrice = (SELECT MAX(UnitPrice) FROM Products p2
                    WHERE p2.CategoryID = p1.CategoryID)
ORDER BY CategoryID;
```

c9-07.sql

The subquery refers to the main query table

CategoryID	ProductID	ProductName	UnitPrice
1	38	Côte de Blaye	263.5
2	63	Vegie-spread	43.9
3	20	Sir Rodney's Marmalade	81
4	59	Raclette Courdavault	55
5	56	Gnocchi di nonna Alice	38
6	29	Thüringer Rostbratwurst	123.79
7	51	Manjimup Dried Apples	53
8	18	Carnarvon Tigers	62.5

# Correlated Subquery Example

---

## ► Explanation to the query

- The subquery references the table in the main query, so the subquery cannot run on its own

## ► For each row to be processed by the main query, the following happens:

- The CategoryID from that row is brought into the subquery
- The subquery finds the maximum UnitPrice for that particular category and returns that value to the main query
- The main query compares the UnitPrice of the row to the value returned from the subquery



# Contents

---

- ▶ Subqueries
- ▶ Correlated Subqueries

## EXISTS





# The EXISTS Operator

---

- ▶ **The operators EXISTS and NOT EXISTS are Boolean operators**
  - Used with subqueries in the WHERE clause
  - Most commonly used with correlated subqueries
- ▶ **The EXISTS operator**
  - Returns *true* if the subquery returns one or more rows
  - Returns *false* if the subquery returns no rows
- ▶ **The NOT EXISTS operator**
  - Returns *true* if the subquery returns no rows
  - Returns *false* if the subquery returns one or more rows

# EXISTS Example

- The Multirow subquery example used NOT IN to find the conference rooms that have no reservations

c9-08.sql

- NOT EXISTS provides an alternative solution

```
SELECT RoomID, Name
FROM   ConferenceRooms cr
WHERE  NOT EXISTS (SELECT 'x' FROM Reservations r
                   WHERE r.RoomID = cr.RoomID);
```

The EXISTS operator does not care about row content, so any expression will do

RoomID	Name
-----	-----
225	2nd Floor Small room
310	3rd Floor Tiny room

The subquery refers to the main query table

# EXISTS and Column Values

---

- ▶ **EXISTS and NOT EXISTS check only for *row existence*, not for *column content* in the subquery rows**
  - The contents of the rows are not relevant
    - So it does not make any difference what column values are returned
    - But the SELECT syntax requires an expression anyway
  - To avoid checking for valid column names, constant values are commonly used
    - SELECT 1, SELECT 0, SELECT 'x', SELECT NULL, etc.
- ▶ **The query would work if any column were selected**
  - Selecting real columns would require more dictionary checking and is not recommended

# EXISTS/NOT EXISTS vs. IN/NOT IN

- ▶ EXISTS and IN are often alternative solutions
- ▶ NOT EXISTS and NOT IN are often alternative solutions
- ▶ NOT IN is always false if there are NULL values within the list
  - One single NULL value is enough to cause the condition to be false
  - Test for IS NOT NULL in the subquery to avoid this problem
  - Alternatively, use NOT EXISTS
- ▶ **Example: The following query gives the same result as the NOT IN and NULL query**

```
SELECT EmployeeID
       , LastName
FROM   Employees e
WHERE  NOT EXISTS
       (SELECT 'x' FROM Employees r
        WHERE r.ReportsTo = e.EmployeeID) ;
```

Different alias names  
to distinguish  
between the two  
tables

c9-09.sql

## Hands-On Exercise 9.2

**In your Exercise Manual, please refer to Hands-On Exercise 9.2: Correlated Subqueries**

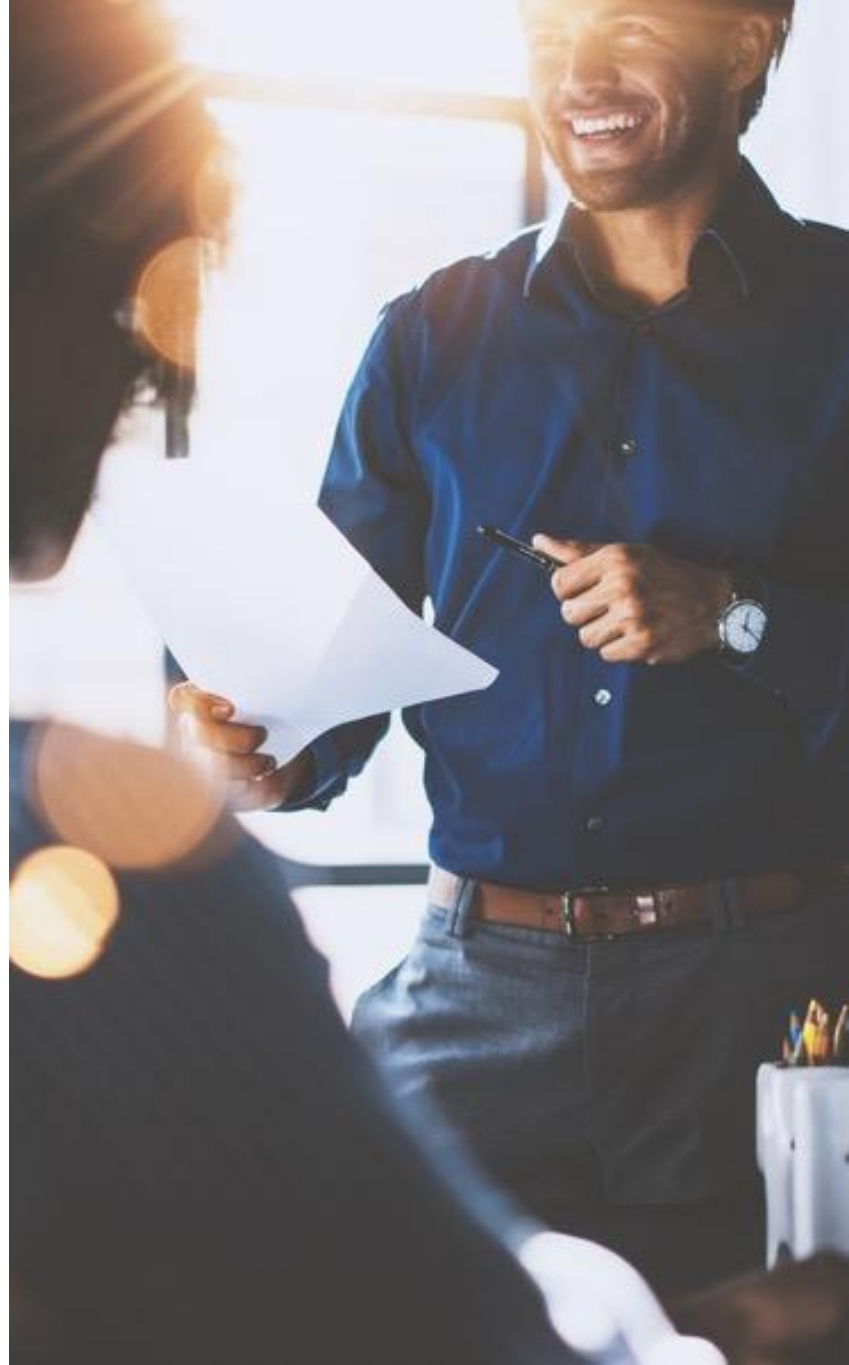
- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**



# Objectives

---

- ▶ Learn the difference between simple and correlated subqueries
- ▶ Use subqueries in conditions and expressions
- ▶ Use the EXISTS operator with subqueries





# Chapter 10

## Views



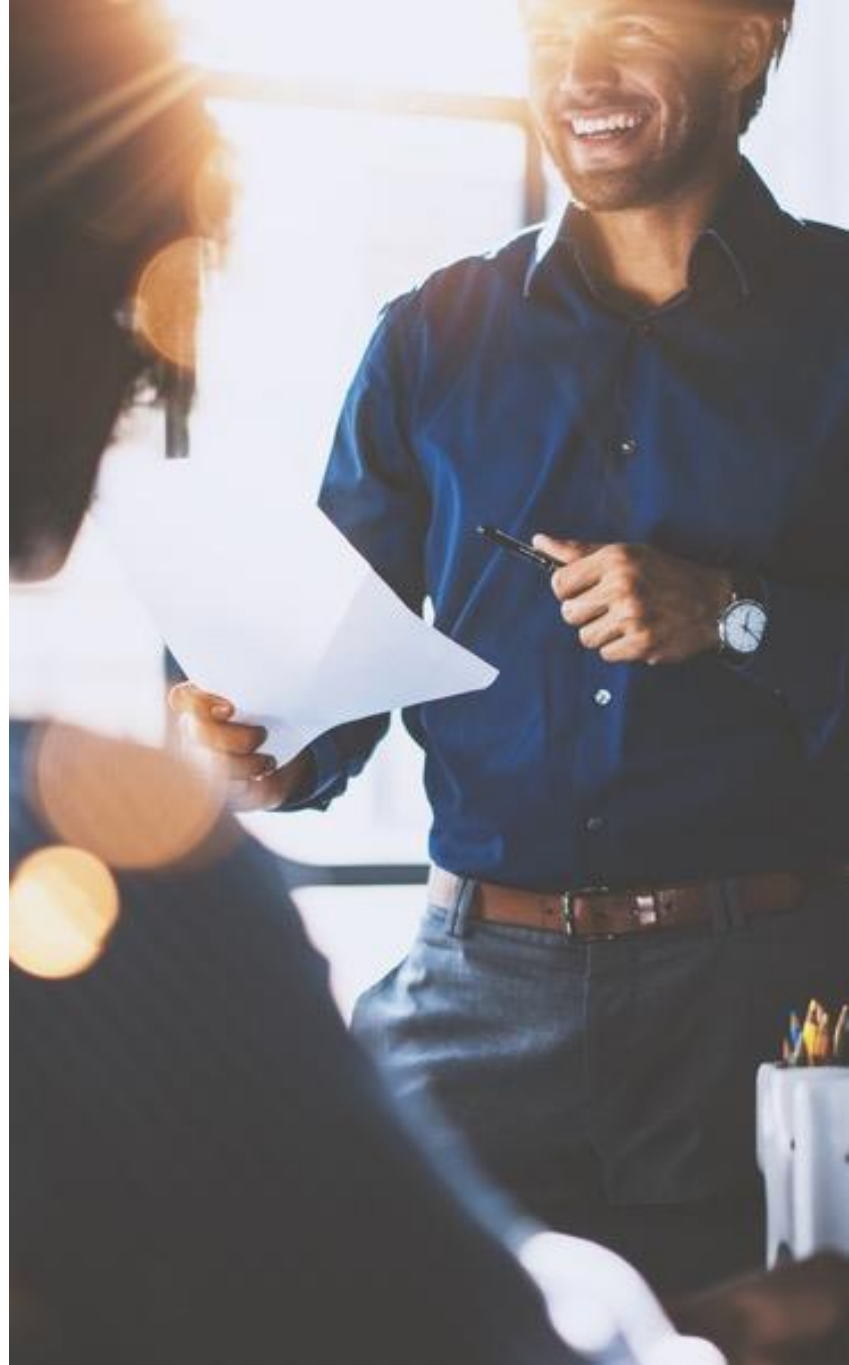
LEARNING TREE™  
INTERNATIONAL



# Objectives

---

- ▶ Learn the difference between inline views and stored views
- ▶ Apply views as reusable code



# Contents

---

## Inline Views

### ► Stored Views



# Views



- ▶ **The result of a SELECT statement always has the structure of a table**
  - It has rows and columns
  - Even if it has only one row and/or one column, it can still be seen as a table
  - We can pretend that the result set actually *is* a table and do SELECT from it
- ▶ **Two basic flavors**
  - A subquery instead of a table name in the FROM clause
    - This is called an *inline* view or *derived table*
  - A named query stored in the system catalog
    - This is called a *stored view*

# Inline View Usage Notes

---

- ▶ **A subquery used as an inline view can be multi-column and multi-row**
  - Just like a real table
- ▶ **A subquery used as an inline view must be self-contained**
  - It *cannot* be correlated
  - It is executed on its own and produces a result set that the main query can read
- ▶ **A subquery used as an inline view must return proper column names**
  - Column alias is required for columns that are expressions
- ▶ **A subquery used as an inline view must be given a table alias to be used as column prefix in the main query**
  - Oracle does not require the alias if it is not referenced as prefix

# Inline View Example

Do Now

## ► Query example that compares rows and aggregates

- The aggregate subquery is repeated three times

c9-06.sql

```
SELECT LastName, CurrentSalary
      ,(SELECT AVG(CurrentSalary) FROM Employees) AS Average
      ,CurrentSalary-(SELECT AVG(CurrentSalary) FROM Employees)
                                     AS Difference
FROM   Employees
WHERE  CurrentSalary<(SELECT AVG(CurrentSalary) FROM Employees);
```

LastName	CurrentSalary	Average	Difference
Davolio	2000	4533.3333	-2533.3333
Peacock	3000	4533.3333	-1533.3333
King	3000	4533.3333	-1533.3333
Suyama	3600	4533.3333	-933.3333
Callahan	4000	4533.3333	-533.3333

# Inline View Example

Do Now

- ▶ **As an alternative, pretend that the result of the subquery actually is a table**

- Make it an inline view
- Use a column alias to give the column a proper name

c10-01.sql

```
SELECT AVG(CurrentSalary) AS AvgSalary FROM Employees;
```

```
AvgSalary  
-----  
4533.3333
```

Pretend that  
this is a table



# Inline View Example

Do Now

## ► Join the inline view to the Employees table

- Use CROSS JOIN because the view has only one row
  - There is no join condition

c10-02.sql

```
SELECT e.LastName, e.CurrentSalary
      ,a.AvgSalary AS Average
      ,e.CurrentSalary-a.AvgSalary AS Difference
FROM   Employees e
CROSS JOIN
      (SELECT AVG(CurrentSalary) AS AvgSalary
       FROM Employees) a
WHERE  e.CurrentSalary<a.AvgSalary;
```

Join to the  
"pretend" table

A view can have an alias  
name, just like a table

LastName	CurrentSalary	Average	Difference
Davolio	2000	4533.3333	-2533.3333
Peacock	3000	4533.3333	-1533.3333
King	3000	4533.3333	-1533.3333
Suyama	3600	4533.3333	-933.3333
Callahan	4000	4533.3333	-533.3333



# Another Inline View Example

Do Now

- ▶ The query can be modified in the previous example to compare to the average per division rather than the overall average
- ▶ The inline view must return a row for each division

c10-03.sql

```
SELECT DivisionID
       ,AVG(CurrentSalary) AS AvgSalary
FROM Employees
GROUP BY DivisionID;
```

Pretend that  
this is a table

DivisionID	AvgSalary
1	5500.0000
2	3760.0000



# Another Inline View Example

Do Now

## ► Join the inline view to the Employees table

- Join on the DivisionID column

c10-04.sql

```
SELECT e.DivisionID, e.LastName, e.CurrentSalary
      ,a.AvgSalary AS Average
      ,e.CurrentSalary-a.AvgSalary AS Difference
FROM   Employees e
JOIN   (SELECT DivisionID, AVG(CurrentSalary) AS AvgSalary
      FROM   Employees
      GROUP BY DivisionID) a
ON     e.DivisionID=a.DivisionID
WHERE  e.CurrentSalary<a.AvgSalary;
```

Join to the  
"pretend" table

DivisionID	LastName	CurrentSalary	Average	Difference
1	Davolio	2000	5500	-3500
1	Buchanan	5000	5500	-500
2	Peacock	3000	3760	-760
2	King	3000	3760	-760
2	Suyama	3600	3760	-160

# Nested Aggregates

- ▶ **Because they alter the structure of the result set, aggregate functions may not be nested**
  - Workaround: Use inline view to achieve multiple levels of aggregation
- ▶ **Example: Find the maximum number of employees in a division**
  - First calculate the number of employees per division

```
SELECT DivisionID, COUNT(*) AS Emps
FROM   Employees GROUP BY DivisionID;
```

c10-05.sql

DivisionID	Emps
1	4
2	5

Pretend that  
this is a table

- Then calculate the maximum from this result set

```
SELECT MAX(Emps) Maxemps
FROM   (SELECT DivisionID, COUNT(*) AS Emps
        FROM Employees GROUP BY DivisionID) d;
```

Maxemps
5

Select from the  
“pretend” table

# Contents

---

## ► Inline Views

## Stored Views



# Stored Views

---

- ▶ **A stored view is a SELECT statement stored in the system catalog**
  - Stored as source text
  - Has a name
- ▶ **When doing SELECT from the view name, the view source text is fetched from the system catalog and expanded into an inline view**
  - Similar to a macro
- ▶ **Syntax for creating a stored view**

```
CREATE VIEW <viewname> [(<column name>, ..., <column name>)]  
AS  
<SELECT statement>;
```
- ▶ **The list of column names is optional**
  - Overrides the column names from the query

# Stored Views

---

- ▶ **The view must return valid column names**
  - Either use column aliases for computed columns (preferred)
  - Or specify column names in the optional column list of the `CREATE VIEW` statement
    - Not recommended; it is easier to read the view definition if column names are taken directly from the query
- ▶ **According to the standard, the `SELECT` statement that defines the view may not do `ORDER BY`**
  - PostgreSQL and Oracle allow this anyway
  - SQL Server allows this if the view does `TOP`
- ▶ **It is permitted to do `ORDER BY` when `SELECT`-ing from a view**

# Stored View Example

Do Now

## ► Reference of query example

c8-07.sql

```
SELECT DivisionID
       ,DepartmentID
       ,COUNT(*) AS Employees
       ,SUM(CurrentSalary) AS SumSal
FROM   Employees
GROUP BY DivisionID, DepartmentID
ORDER BY DivisionID, DepartmentID;
```

DivisionID	DepartmentID	Employees	SumSal
1	100	2	11000
1	200	2	11000
2	200	1	3000
2	300	4	15800



# Stored View Example

Do Now

## ► Create a view based on the query example

- Without ORDER BY

```
CREATE VIEW EmpStatistics AS
SELECT DivisionID
       ,DepartmentID
       ,COUNT(*) AS Employees
       ,SUM(CurrentSalary) AS SumSal
FROM   Employees
GROUP BY DivisionID, DepartmentID;
```

c10-06.sql

## ► Then SELECT from the view

```
SELECT * FROM EmpStatistics
ORDER BY DivisionID, DepartmentID;
```

How do you know that  
this is a view and not a  
real table?

DivisionID	DepartmentID	Employees	SumSal
1	100	2	11000
1	200	2	11000
2	200	1	3000
2	300	4	15800

# Stored Views in the System Catalog

---

- ▶ **To see the view definition in the system catalog**

- ▶ **PostgreSQL and SQL Server**

```
SELECT * FROM INFORMATION_SCHEMA.views  
WHERE table_name = '<name>';
```

- ▶ **Oracle**

```
SELECT * FROM USER_VIEWS  
WHERE VIEW_NAME = '<name>';
```

- ▶ **SQL Server can also do**

```
EXEC sp_helptext '<name>';
```

- ▶ **Many client GUI tools can display view definitions**

# Updating Through a View

---

- ▶ **A view does not store data content**
- ▶ **If you do INSERT, UPDATE, or DELETE on view, the data values in the underlying table(s) will be updated**
  - This is not commonly done
- ▶ **Updating is not possible on all views**
  - General Rule: A view is updatable if the update can be deterministically mapped back to the underlying table(s)
- ▶ **Computed columns may not be updated**
- ▶ **Updates are not permitted on aggregate views or views with set operators or DISTINCT**
- ▶ **Join views may or may not be updatable, depending on product and version**

# Dropping and Recreating Views

---

- ▶ **A view does not store data content**
  - When dropping a view, the underlying table(s) are still intact
- ▶ **To change a view definition, the view can be dropped and recreated**
- ▶ **To change a view definition without dropping it**
  - In PostgreSQL and Oracle, do `CREATE OR REPLACE VIEW`
  - In SQL Server, do `ALTER VIEW`

# Advantages of Stored Views

---

## ► Modularity and simplification

- Reusable code: Solve a problem once and reuse the solution
- It is possible to build views upon views
  - A SELECT statement that reads from a view may be created as a view

## ► Modular data independence

- If you need to change the underlying table structure, you may be able to build views that look like the old structure
  - This reduces the need for modifying application programs

## ► Security and access control

- To access a view, a user needs privileges on the view, not on the underlying table(s)
  - A view can serve as a “filter” to limit access to parts of a table
- The view can
  - Contain only some of the table columns
  - Have a WHERE filter that limits row access
  - Be an aggregate that hides single-row details



# Hands-On Exercise

## 10.1

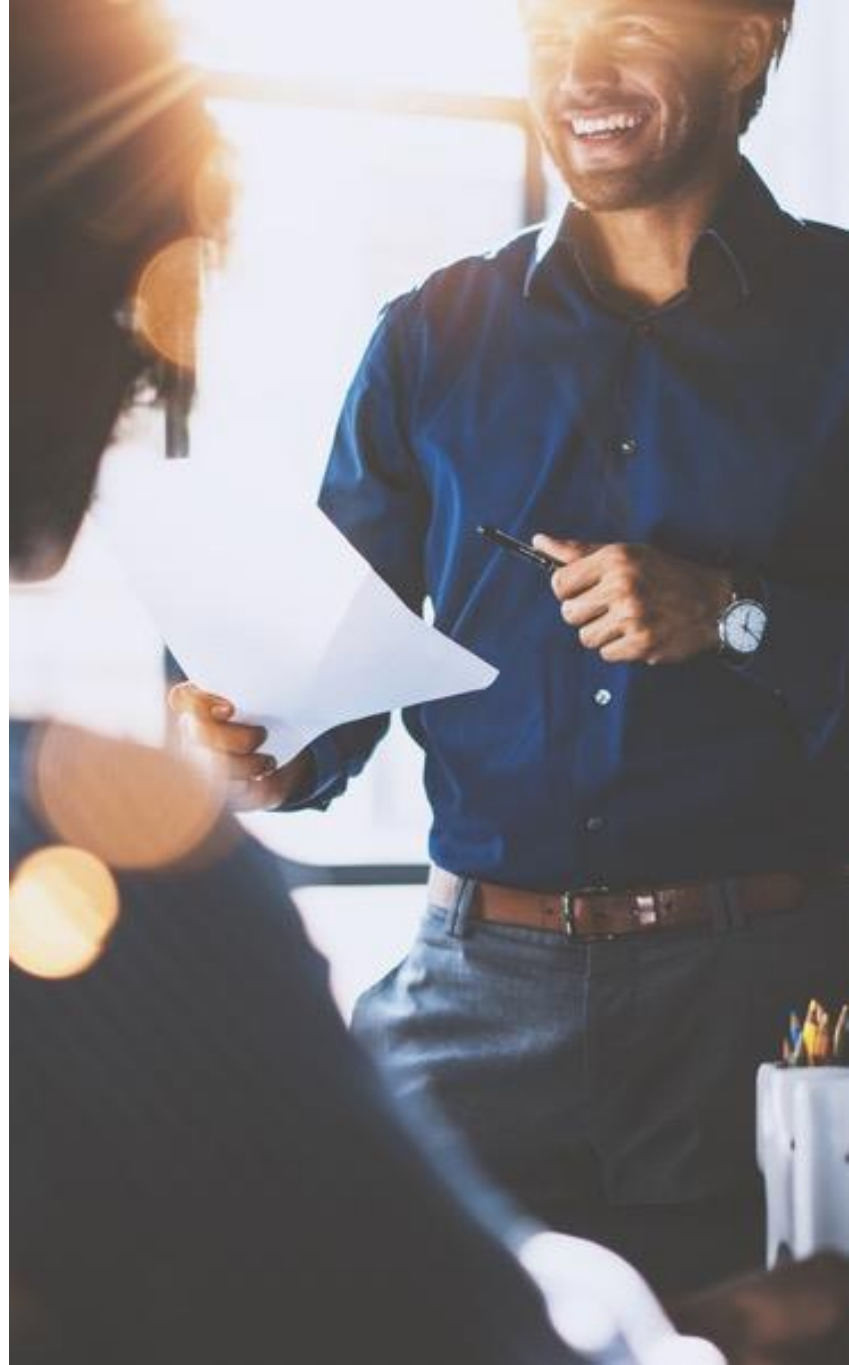
**In your Exercise Manual, please refer to  
Hands-On Exercise 10.1: Views**

- ▶ **Solutions to all the exercises are provided as files located in the folder `C:\Course925Solutions`**
- ▶ **You should save your own solutions to the exercises, since some of the exercises build on previous exercises**
- ▶ **However, if you have not saved your own solutions, you can always copy from the provided solution files**

# Objectives

---

- ▶ Learn the difference between inline views and stored views
- ▶ Apply views as reusable code





# Chapter 11

## Course Summary

# Course Objectives

---

- ▶ **Use SQL to define and modify database structures**
  - CREATE, ALTER, DROP
- ▶ **Use SQL to update database contents**
  - INSERT, UPDATE, DELETE
  - Transactions, COMMIT, ROLLBACK
- ▶ **Query database content**
  - Simple SELECT
  - Joins
  - Functions
  - Aggregates
  - Subqueries
  - Views

